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

Overview

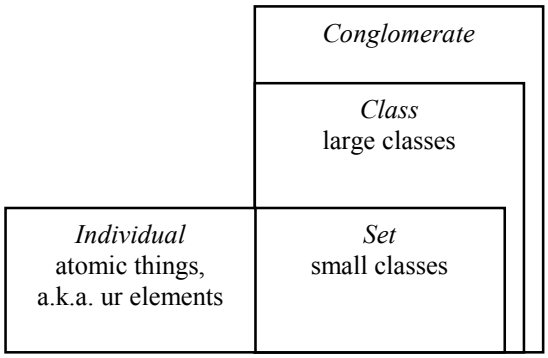

Diagram 1: Collection Hierarchy

The *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* (1990) by Jiří Adámek, Horst Herrlich and George E. Strecker. 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. The Foundation Ontology is made up of several namespaces: the Namespace of Large Sets, the Namespace of Large Relations and the Namespace of Orders, the Namespace of Large Classifications, the Namespace of Large Concept Lattices, etc. There is a foundation restriction at the lower metalevel – the *Set Namespace* (for small sets and their functions), the *Relation Namespace* and the *Classification & Concept Lattice Namespaces* are each restrictions of their large counterparts.

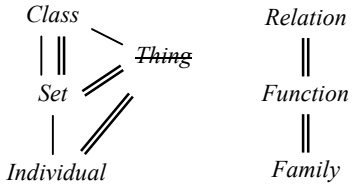

Figure 1: Collection Hierarchies

The Foundation Ontology uses and imports the following terms from the Basic KIF Ontology.

KIF	‘class’, ‘relation’, ‘subclass’, ‘function’
	‘signature’
	‘[-]’, ‘1’, ‘2’

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’, and the generic type declaration term ‘KIF$signature’. The term ‘KIF$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 here to *conglomerate*.

The IFF Foundation Ontology uses the three-level set-theoretic hierarchy of sets – classes – conglomerates. In the following table we located various collections and functions[1] in this 3-tiered framework. The placement of morphisms uses the axiom of replacement: there is no surjection from a set to a proper class.

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
Sets	○	sets as collections and the extent of (set) functions – the objects and morphisms in the Set category
	○	the instance and type sets of the classifications and the instance and type functions of the infomorphisms in the Classification category
	○	any (small) relation regarded as a collection – objects in the Relation category

[1] 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.

Previous Foundations

Topos Axioms

The 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 Foundation Ontology will be proven to be a well-pointed quasi-topos with natural numbers and choice (for its small restriction, just remove the 'quasi-' prefix). Perhaps the Foundation Ontology would partially satisfy 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. Connections with the Foundation Ontology are indicated.

Category Axioms

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

- 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 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$16`].
- Given two objects A and B in a category the *exponent* '`B^A`' is the collection of all morphisms from A to B. In the Foundation Ontology the exponent class is declared by the term '`SET.CCC$exponent`' [axiom `SET.CCC$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.CCC$evaluation`' [axiom `SET.CCC$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 $0 = \textit{false}$ and $1 = \textit{true}$. 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.CCC$adjoint`' in axiom `SET.CCC$3`.

10. An *elementary topos* is a Cartesian-closed category that has a subobject classifier – an object Ω and a morphism $\textit{true} : 1 \to \Omega$ 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 below (Figure 3) 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`.

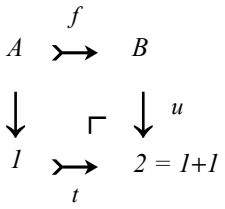

Figure 2: Subobject Classifier

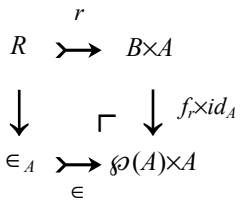

Figure 3: Power Objects

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

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

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

Classical Analysis

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

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

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

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

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

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

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

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

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

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

Sketches

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

A sketch is the categorical counterpart of a first-order theory. It specifies the language of the theory in terms of limits and colimits of diagrams. The language of (finitary) quantifier-free logic is representable entirely with finite product (FP) sketches, i.e. no colimits and only discrete limits. Finite

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

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

Fibrations

There is a need to incorporate some aspect of fibrations and indexed categories in the Foundation Ontology. For one example, the (small) Classification Namespace represents the category Classification, which is part of a fibered span (see Figure 2). 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.

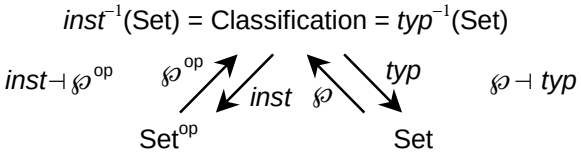

Figure 4: The Classification Fibered Span

Part I: The Large Aspect

The Namespace of Conglomerates

In a set-theoretic sense, this namespace sits at the top of the IFF Foundation Ontology. The suggested prefix for this namespace is '`CNG`', standing for conglomerates. When used in an external namespace, all terms that originate from this namespace can be prefixed with '`CNG`'. This namespace represents conglomerates, and their functions and relations. No sub-namespaces are needed. As illustrated in Diagram 1, conglomerates characterized the overall architecture for the large aspect of the Foundation Ontology. Nodes in this diagram represent conglomerates and arrows represent conglomerate functions. The small oval on the right,

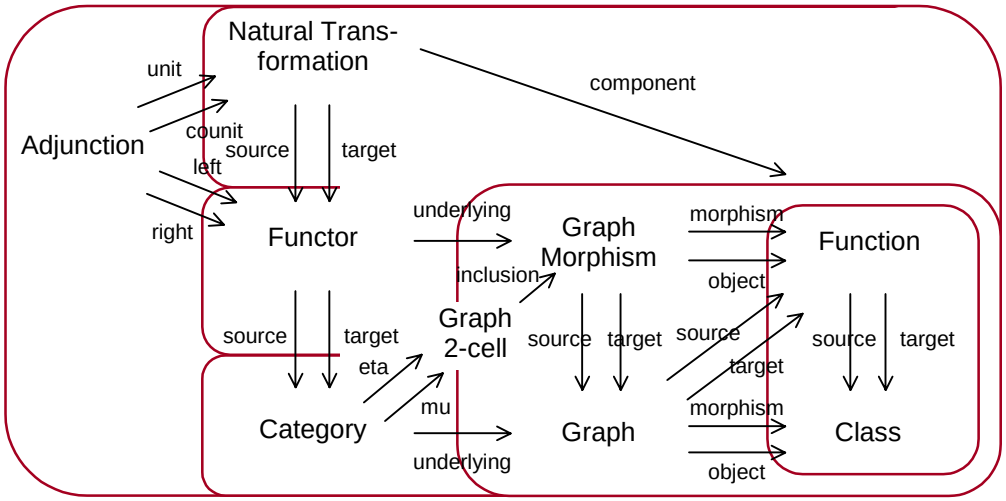

Diagram 2: Core Conglomerates and Functions

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.

Conglomerates

`CNG`

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

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

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

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

- There is a *subconglomerate* binary KIF relation.

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

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

- There is a *disjoint* binary KIF relation.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The Namespace of Classes (Large Sets)

This is the core namespace in the Foundation Ontology. The suggested prefix for this namespace is 'SET', standing for large sets. When used in an external namespace, all terms that originate from this namespace can be prefixed with 'SET'. This namespace represents classes (large sets) and their functions. The following terms are declared and axiomatized in this namespace. As indicated in the left column of Table 2, several sub-namespaces are needed.

Table 2: Terms introduced in the large set namespace

	KIF$class	Unary KIF$function	Binary KIF$function
SET	'class'		
SET .FTN	'function' 'parallel-pair' 'injection', 'surjection', 'bijection' 'monomorphism', 'epimorphism', 'isomorphism'	'source', 'target', 'identity' 'image', 'inclusion', 'fiber', 'inverse-image'	'composition'
SET .LIM	'unit', 'terminal'	'unique' 'tau-cone', 'tau'	
SET .LIM .PRD	'diagram', 'pair' 'cone'	'class1', 'class2', 'opposite' 'cone-diagram', 'vertex', 'first', 'second' 'limiting-cone', 'limit', 'binary-product', 'projection1, 'projection2' 'mediator' 'binary-product-opspan' 'tau-cone', 'tau'	'pairing-cone', 'pairing'
SET .LIM .PRD .FTN	'pair'	'source', 'target', 'class1', 'class2' 'binary-product'	
SET .LIM .EQU	'diagram', 'parallel-pair', 'cone'	'source', 'target', 'function1', 'function2' 'cone-diagram', 'vertex', 'function' 'limiting-cone', 'limit', 'equalizer', 'canon' 'mediator' 'kernel-parallel-pair', '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-opspan', 'kernel-pair' 'tau-cone', 'tau'	'pairing-cone', 'pairing'
SET .CCC		'evaluation' 'adjoint'	'exponent'
SET .TOP		'subclass' 'element', 'el2ftn' 'truth', 'true' 'character'	'constant'

The signatures of some of the relations and functions in the CNG and SET namespace are as follows.

Table 3: Signatures for some relations and functions in the large set and conglomerate namespaces

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

Classes

SET

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

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

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

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

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

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

- There is a *disjoint* binary CNG relation.

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

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

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

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

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

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

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

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

Functions

`SET.FTN`

A (class) function 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 func-

tions, 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 (Figure 1) $f : X \to Y$ 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.

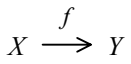

Figure 5: Class Function

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 (Figure 6) with the source/target inclusions. Ordinary restriction is a constraint on the conglomerate function – it says that on the class function source subcollection the conglomerate function maps into the class function target subcollection.

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

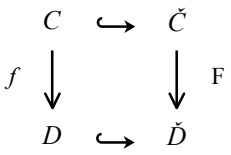

Figure 6: Ordinary restriction

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

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

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

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

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

…

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

```
(8) (CNG$conglomerate endofunction)
    (forall (?f (endofunction ?f))
```

```
        (and (function ?f)
             (= (source ?f) (target ?f))))
```

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

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

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

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

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

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

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

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

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

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

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

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

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

- Composition satisfies the usual *associative law*.

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

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

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

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

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

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

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

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

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

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

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

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

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

```
(= injection monomorphism)
```

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

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

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

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

```
(= surjection epimorphism)
```

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

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

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

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

```
(= bijection isomorphism)
```

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

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

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

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

Finite Limits

SET.LIM

Here we present axioms that make the quasi-category of classes and functions finitely complete. We assert the existence of terminal classes, binary products, equalizers of parallel pairs of functions and pullbacks of opspans. All are defined to be specific classes – for example, the binary product is the Cartesian product.

Because of commonality, the terminology for binary product, equalizer, subequalizer and pullbacks are put into sub-namespace. The *diagrams* and *limits* are denoted by both generic and specific terminology.

The Terminal Class

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

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

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

Binary Products

`SET.LIM.PRD`

An *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 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)))))
```

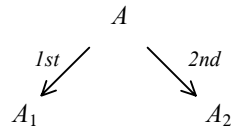

Figure 7: Product Cone

- There is a unary CNG function '`limiting-cone`' that maps a pair (of classes) to its binary product (limiting binary product cone). 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))))
```

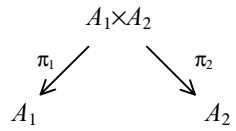

Figure 8: Limit Cone

- 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. This is a specific *Cartesian Binary Product* function. Let '`pair`' be the SET Namespace term that denotes the function pair collection.

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

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

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

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

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

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

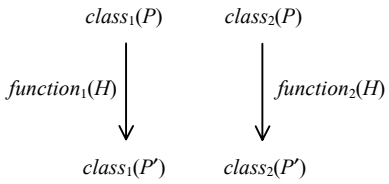

Figure 8: Pair Morphism

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 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)))))
```

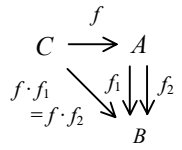

Figure 9: Equalizer Cone

- There is a unary CNG function '`limiting-cone`' that maps a parallel-pair to its equalizer (limiting equalizer cone). 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))))
```

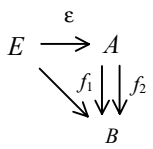

Figure 10: Limit Cone

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

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

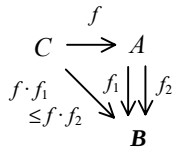

Figure 11: Subequalizer Cone

- 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). 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))))

(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)))
```

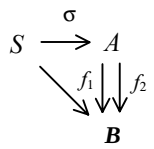

Figure 12: Lax Limit Cone

- 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 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.

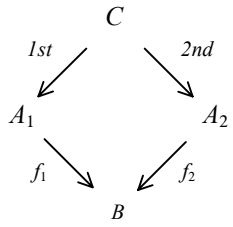

Figure 13: Pullback Cone

```
(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). 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.

```
(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)
```

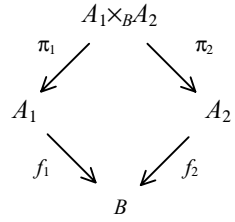

Figure 14: Limit Cone

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

- 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, 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,

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

and two projection functionals

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

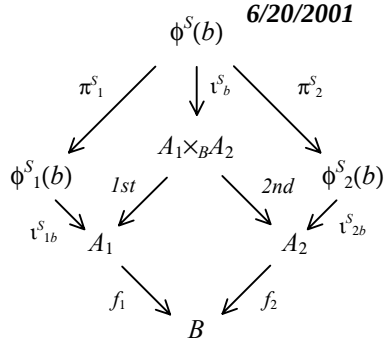

Figure 15: Pullback Fibers

Here are the pointwise definitions.

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

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

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

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

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

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

(24) (CNG$function fiber2)
     (CNG$signature fiber2 opspan SET.FTN$function)
     (forall (?s (opspan ?s))
        (and (= (SET.FTN$source (fiber2 ?s)) (opvertex ?s))
             (= (SET.FTN$target (fiber2 ?s)) (SET$power (class2 ?s)))
             (= (fiber2 ?s) (SET.FTN$fiber (opsecond ?s))))))

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

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

(27) (KIF$function fiber-embedding)
     (KIF$signature fiber-embedding opspan CNG$function)
```

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

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

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

    (30) (KIF$function fiber12-embedding)
        (KIF$signature fiber12-embedding opspan CNG$function)
        (forall (?s (opspan ?s))
           (CNG$signature (fiber12-embedding ?s) (class1 ?s) SET.FTN$function))
        (forall (?s (opspan ?s)
                 ?x1 ((class1 ?s) ?x1))
           (and (= (SET.FTN$source ((fiber12-embedding ?s) ?x1))
                   ((fiber12 ?s) ?x1))
                (= (SET.FTN$target ((fiber12-embedding ?s) ?x1))
                   ((fiber ?s) ((opfirst ?s) ?x1)))))
        (forall (?s (opspan ?s)
                ?x1 ((class1 ?s) ?x1)
                ?x2 (((fiber12 ?s) ?x1) ?x2))
            (= (((fiber12-embedding ?s) ?x1) ?x2) [?x1 ?x2]))

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

    (32) (KIF$function fiber1-projection)
```

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

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

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

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

    (35) (CNG$function kernel-pair)
        (CNG$signature kernel-pair SET.FTN$function equivalence-relation)
        (forall (?f (SET.FTN$function ?f))
           (and (= (REL$class (kernel-pair ?f)) (source ?f))
                (= (REL$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 are required to have a common target class, denoted as the *opvertex*. Let '`opspan`' be the SET Namespace term that denotes the *Opspan* 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)
```

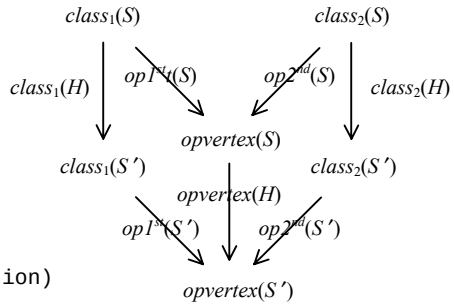

Figure 16: Opspan Morphism

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

Cartesian Closure

SET.CCC

- 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))))))
```

Topos Structure

`SET.TOP`

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

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

```
(1) (KIF$function subobject)
    (KIF$signature subobject SET$class SET$class)
    (forall (?c (SET$class ?c) ?f)
       (<=> ((subobject ?c) ?f)
            (and (SET.FTN$injection ?f)
                 (= (SET.FTN$target ?f) ?c))))
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

    (forall (?c (SET$class ?c)
             ?x ((element ?c) ?x)
```

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

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

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

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

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

The Namespace of Large Relations

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

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

Relations

REL

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

$$R \subseteq X_1 \times X_2$$

Figure 17: Large Binary Relation

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Endorelations

REL.ENDO

- Endorelations are special relations.

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

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

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

- The is a *subendorelation* relation.

```
(4) (CNG$relation subendorelation)
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
(10) (CNG$conglomerate transitive)
     (CNG$subconglomerate transitive endorelation)
     (forall (?r (endorelation ?r))
         (<=> (transitive ?r)
              (forall (?x1 ((class ?r) ?x1)
                       ?x2 ((class ?r) ?x2)
                       ?x3 ((class ?r) ?x3))
```

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

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

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

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

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

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

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

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

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

The Namespace of Large Orders

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

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

Orders

ORD

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

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

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

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

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

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

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

The Namespace of Large Graphs

This is all finished.

The Namespace of Categories

This is mostly finished.

The Namespace of Functors

This is partly finished.

The Namespace of Natural Transformations

The Namespace of Adjunctions

The Namespace of Large Classifications

The Namespace of Large Concept Lattices

Part II: The Small Aspect

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.