# The Core Ontology



## The Namespace of Conglomerates

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

### *Conglomerates*

**`CNG`**

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

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

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

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

- There is a *subconglomerate* binary KIF relation.

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

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

- There is a *disjoint* binary KIF relation.

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

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

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

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

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

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

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

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

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

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

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

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

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

(10) (KIF$function extent)
     (KIF$signature extent relation conglomerate)
     (forall (?r (relation ?r))
         (and (subconglomerate
                  (extent ?r)
                  (binary-product (conglomerate1 ?r) (conglomerate2 ?r)))
              (forall (?x1 ((conglomerate1 ?r) ?x1)
                       ?x2 ((conglomerate2 ?r) ?x2))
                  (<=> ((extent ?r) [?x1 ?x2])
                       (?r ?x1 ?x2))))))
```

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

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

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

## The Namespace of Classes (Large Sets)

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

**Table 1: Terms introduced in the core namespace**

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

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

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

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

| | | |
|---|---|---|
| *subconglomerate*<br>*⊆ conglomerate × conglomerate*<br>*signature ⊆ function × KIF$sequence*<br>*subclass ⊆ class × class*<br>*disjoint ⊆ class × class*<br>*partition ⊆ class × KIF$sequence*<br>*restriction ⊆ function × CNG$function*<br>*restriction-*<br>*pullback ⊆ function × CNG$function* | *source, target* : *function → class*<br>*identity, range* : *function → class*<br>*vertex* : *span → class*<br>*first*, *second* : *span → function*<br>*opvertex* : *opspan → class*<br>*opfirst*, *opsecond* : *opspan → function*<br>*opposite* : *opspan → opspan* | *composition* : *function × function → function* |
| | *unique* : *class → function*<br>*cone-opspan* : *cone → opspan*<br>*vertex* : *cone → class*<br>*first*, *second*, *mediator* : *cone → function*<br>*limiting-cone* : *opspan → cone* | *binary-product* : *class × class → class*<br>*binary-product-*<br>*opspan* : *class × class → opspan* |
| | *power* : *class → relation* | *exponent* : *class × class → class*<br>*evaluation* : *class × class → function* |

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

**Table 4: Correspondence between Mathematical Notation and Ontological Terminology**

| Math | Ontological Terminology | Natural Language Description |
|---|---|---|
| ⊆ | ‘SET$subclass’ | the subclass inclusion relation |
| ≅ | | the isomorphism relation between objects |
| ∅ | ‘SET.LIM$null’,<br>‘SET.LIM$initial’ | the empty class – this is the initial object in the quasi-category of classes and functions |
| × | ‘SET.LIM.PRD$binary-product’ | binary product operator on objects |

## *Classes*

### SET

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

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

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

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

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

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

- There is a *disjoint* binary CNG relation on classes.

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

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

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

- For any pair of classes there is a binary union class and a binary intersection class.

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

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

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

class *X* the *power-class* over *X* is the collection of all subclasses of *X*. There is a unary CNG '`power`' function that maps a class to its associated power.

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

## *Functions*

**SET.FTN**

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

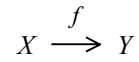


**Figure 1: Class Function**

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

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

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

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

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

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

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

  - Any function can be embedded as a binary relation.

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

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

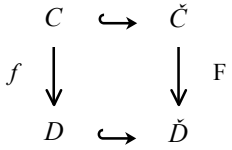


**Diagram 1: Ordinary restriction**

```
(5) (KIF$relation restriction)
    (KIF$signature restriction function CNG$function)
    (forall (?ftn ?FTN (function ?ftn) (CNG$function ?FTN))
        (<=> (restriction ?ftn ?FTN)
```

```
        (exists (?cng1 (CNG$conglomerate ?cng1)
                ?cng2 (CNG$conglomerate ?cng2))
            (and (CNG$signature ?FTN ?cng1 ?cng2)
                 (CNG$subconglomerate (source ?ftn) ?cng1)
                 (CNG$subconglomerate (target ?ftn) ?cng2)
                 (forall (?x ((source ?ftn) ?x))
                     (= (?ftn ?x) (?FTN ?x))))))
```

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

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

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

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

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

        ...
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- Composition satisfies the usual *associative law*.

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

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

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

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

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

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

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

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

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

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

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

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

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

```
(= injection monomorphism)
```

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

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

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

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

```
(= surjection epimorphism)
```

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

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

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

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

```
(= bijection isomorphism)
```

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

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

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

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

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

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

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

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

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

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

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

$$f_{@}(a, b) \text{ iff } a \leq_{\boldsymbol{A}} f(b),$$

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

$$f^{@}(b, a) \text{ iff } f(b) \leq_{\boldsymbol{A}} a.$$

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

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

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

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

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

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

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

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

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

## *Finite Completeness*

**SET.LIM**

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

### The Terminal Class

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

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

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

### Binary Products

**SET.LIM.PRD**

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

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

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

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

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

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

- Every pair has an opposite.

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

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

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

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

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

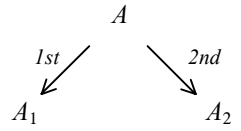


**Figure 2: Product Cone**

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

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

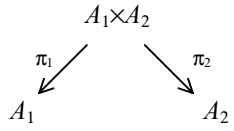


**Figure 3: Limiting Cone**

```
(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))))
```

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

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

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

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

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

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

```
            (= (SET.LIM.PBK$opfirst (binary-product-opspan ?p))
               (unique (class1 ?p)))
            (= (SET.LIM.PBK$opsecond (binary-product-opspan ?p))
               (unique (class2 ?p)))))
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## Function

**SET.LIM.PRD.FTN**

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

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

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

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

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

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

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

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

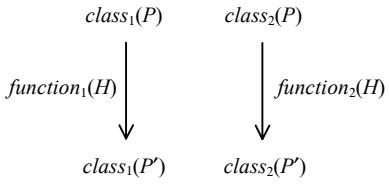


**Figure 4: Function Pair**

## Equalizers

**SET.LIM.EQU**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

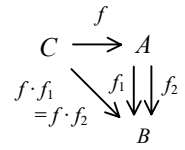


**Figure 5: Equalizer Cone**

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

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

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

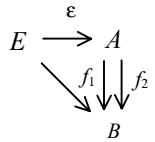


**Figure 6: Limiting Cone**

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

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

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

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

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

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

## Subequalizers

**SET.LIM.SEQU**

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

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

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

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

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

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

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

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

```
            (= (SET.FTN$target (function2 ?p)) (ORD$class (order ?p)))))
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

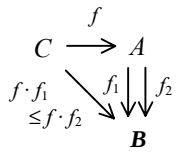


**Figure 7: Subequalizer Cone**

- There is a unary CNG function '`limiting-lax-cone`' that maps a lax-parallel-pair $f_1, f_2 : A \to \boldsymbol{B} = \langle B, \leq \rangle$ to its subequalizer (lax limiting subequalizer cone) (Figure 8). Axiom (*) asserts that this function is total. This, along with the universality of the mediator function, implies that an subequalizer exists for any lax-parallel-pair. The vertex of the subequalizer cone is a specific *Cartesian*

*Subequalizer* class $\{a \in A \mid f_1(a) \leq f_2(a)\} \subseteq A$ given by the CNG function '`subequalizer`'. It comes equipped with a CNG canonical subequalizing function '`subcanon`', which is the inclusion of the subequalizer class into source class $A$. This notation is for convenience of reference. It is used for subequalizers in general. Axiom (#) ensures that this subequalizer is specific – that it is exactly the subclass of the source class on which the two functions are ordered. Obviously, equalizers are a special case of subequalizers – just use the lax embedding of the equalizer diagram.

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

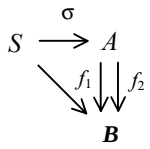


**Figure 8: Lax Limiting Cone**

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

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

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

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

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

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

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

(17) (CNG$function suborder)
     (CNG$signature suborder ORD$order SET$class)
     (forall (?o (ORD$order ?o))
         (= (suborder ?o) (subequalizer (suborder-lax-diagram ?o))))
```

## Pullbacks

**`SET.LIM.PBK`**

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

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

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

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

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

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

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

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

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

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

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

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

- Every opspan has an opposite.

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

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

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

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

- *Pullback cones* are used to specify and axiomatize pullbacks. Each pullback cone (Figure 9) has an underlying *opspan*, a *vertex* class, and a pair of functions called *first* and *second*, whose common source class is the vertex and whose target classes are the source classes of the functions in the opspan. The first and second functions form a commutative diagram with the opspan. A pullback cone is the very

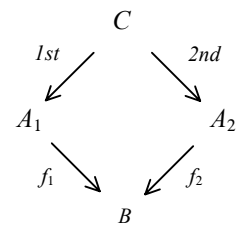


**Figure 9: Pullback Cone**

special case of a cone over an opspan. Let ‘cone’ be the SET namespace term that denotes the *Pullback Cone* collection.

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

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

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

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

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

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

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

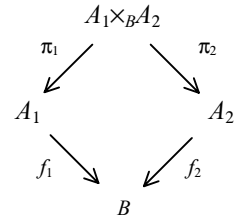


**Figure 10: Limiting Cone**

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

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

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

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

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

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

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

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

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

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

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

$$\phi^S : B \to \wp(A_1 \times_B A_2)$$
$$\phi^S_1 : B \to \wp A_1 \qquad \phi^S_{12} : A_1 \to \wp A_2$$
$$\phi^S_2 : B \to \wp A_2 \qquad \phi^S_{21} : A_2 \to \wp A_1$$

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

$$\iota^S_b : \phi^S(b) \to A_1 \times_B A_2$$
$$\iota^S_{1b} : \phi^S_1(b) \to A_1 \qquad \iota^S_{12a1} : \phi^S_{12}(a_1) = \phi_2{}^S(f_1(a_1)) \to \phi^S(f_1(a_1))$$
$$\iota^S_{2b} : \phi^S_2(b) \to A_2 \qquad \iota^S_{21a2} : \phi^S_{21}(a_2) = \phi_1{}^S(f_2(a_2)) \to \phi^S(f_2(a_2))$$

and two projection functionals

$$\pi^S_{1b} : \phi^S(b) \to \phi^S_1(b)$$
$$\pi^S_{2b} : \phi^S(b) \to \phi^S_2(b)$$

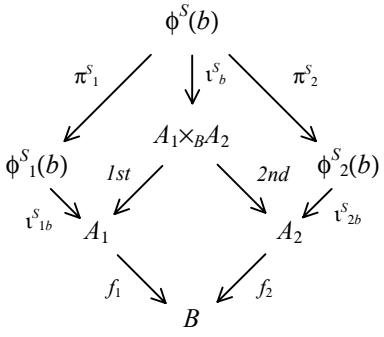


**Diagram 2: Pullback Fibers**

Here are the pointwise definitions.

$$\phi^S(b) = \{(a_1, a_2) \in A_1 \times_B A_2 \mid f_1(a_1) = b = f_2(a_2)\} \subseteq A_1 \times_B A_2$$
$$\phi^S_1(b) = \{a_1 \in A_1 \mid f_1(a_1) = b\} \subseteq A_1 \qquad \phi^S_{12}(a_1) = \{a_2 \in A_2 \mid f_1(a_1) = f_2(a_2)\} = \phi^S_2(f_1(a_1))$$
$$\phi^S_2(b) = \{a_2 \in A_2 \mid b = f_2(a_2)\} \subseteq A_2 \qquad \phi^S_{21}(a_2) = \{a_1 \in A_1 \mid f_1(a_1) = f_2(a_2)\} = \phi^S_1(f_2(a_2))$$

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

$$\phi^S = (I^{st} \cdot f_1)^{-1}$$
$$\phi^S_1 = f_1^{-1} \qquad \phi^S_{12} = f_1 \cdot f_2^{-1}$$
$$\phi^S_2 = f_2^{-1} \qquad \phi^S_{21} = f_2 \cdot f_1^{-1}$$

We clearly have the identifications: $f_1 \cdot \phi^S_2 = \phi^S_{12}$ and $f_2 \cdot \phi^S_1 = \phi^S_{21}$.

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

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

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

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

(26) (CNG$function fiber21)
```

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

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

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

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

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

    (31) (KIF$function fiber21-embedding)
        (KIF$signature fiber21-embedding opspan CNG$function)
        (forall (?s (opspan ?s))
           (CNG$signature (fiber21-embedding ?s) (class2 ?s) SET.FTN$function))
        (forall (?s (opspan ?s)
                ?x2 ((class2 ?s) ?x2))
           (and (= (SET.FTN$source ((fiber21-embedding ?s) ?x2))
                   ((fiber21 ?s) ?x2))
```

```
            (= (SET.FTN$target ((fiber21-embedding ?s) ?x2))
               ((fiber ?s) ((opsecond ?s) ?x2)))))
     (forall (?s (opspan ?s)
             ?x2 ((class2 ?s) ?x2)
             ?x1 (((fiber21 ?s) ?x2) ?x1))
        (= (((fiber21-embedding ?s) ?x2) ?x1) [?x1 ?x2]))

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

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

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

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

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

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

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

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

```
        (= (SET.FTN$target (tau ?s)) (pullback ?s))))
(forall (?s (opspan ?s))
   (= (tau ?s) (mediator (tau-cone ?s))))
```

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

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

## Opspan Morphisms

**SET.LIM.PBK.MOR**

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

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

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

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

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

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

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

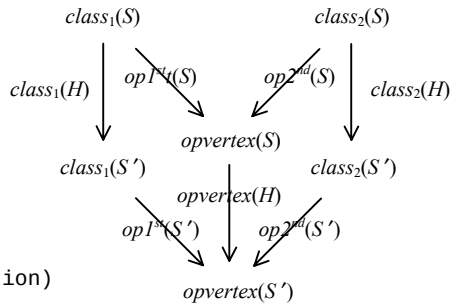


**Figure 11: Opspan Morphism**

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

## *Topos Structure*

**SET.TOP**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## *Finite Cocompleteness*

**SET.COL**

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

## The Namespace of Large Relations

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

**Table 1: Terms introduced into the large relation namespace**

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

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

**Table 2: Correspondence between Mathematical Notation and Ontological Terminology**

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

## *Relations*

**REL**

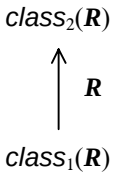


**Figure 1: Large Binary Relation**

A (large) binary relation (Figure 1) is a special case of a conglomerate binary relation with classes for its two coordinates. A class relation is also known as a SET relation. A SET relation is intended to be an abstract semantic notion. Syntactically however, every relation is represented as a binary KIF relation. The signature of SET relations, considered to be CNG relations, is given by their two classes. A SET relation $\boldsymbol{R} = \langle class_1(\boldsymbol{R}), class_2(\boldsymbol{R}), extent(\boldsymbol{R})\rangle$ consists of a *first* class $class_1(\boldsymbol{R})$, a *second* class $class_2(\boldsymbol{R})$, and an *extent* class $extent(\boldsymbol{R}) \subseteq class_1(\boldsymbol{R}) \times class_2(\boldsymbol{R})$. We often use the following morphism notation for binary relations: $\boldsymbol{R} : class_1(\boldsymbol{R}) \to class_2(\boldsymbol{R})$.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- Two relations $R$ and $S$ are *composable* when the target class of $R$ is the same as the source class of $S$. There is a binary CNG function *composition* that takes two composable relations and returns their composition.

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

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

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

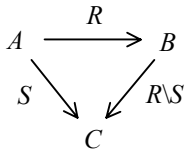


**Figure 3: Left-Residuation**

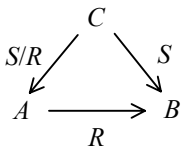


**Figure 4: Right-Residuation**

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

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

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

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

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

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

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

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

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

- Residuation preserves composition

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

  Residuation preserves identity

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

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

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

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

- A theorem about transpose states that transpose dualizes residuation:

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

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

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

- We can prove a general associative law:

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

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

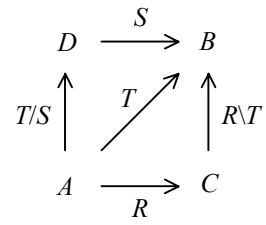


**Figure 5: Associative law**

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

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

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

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

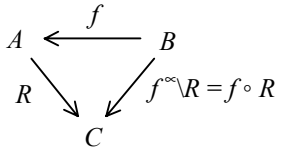


**Figure 6: pre-composition**

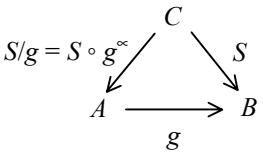


**Figure 7: post-composition**

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

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

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

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

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

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

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

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

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

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

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

## *Endorelations*

**REL.ENDO**

- Endorelations are special relations.

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

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

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

- The is a *subendorelation* relation.

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

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

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

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

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

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

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

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

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

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

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

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

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

- Or expressed more abstractly (without elements).

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

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

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

- Or expressed more abstractly (without elements).

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

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

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

- Or expressed more abstractly (without elements).

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

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

```
(15) (CNG$conglomerate transitive)
     (CNG$subconglomerate transitive endorelation)
     (forall (?r (endorelation ?r))
```

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

- Or expressed more abstractly (without elements).

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

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

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

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

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

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

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

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

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