The IFF Core Ontology

The Namespace of Classes (Large Sets)

This is the main namespace in the Core (sub)Ontology of the IFF Foundation Ontology. This namespace represents classes (large sets) and their functions. 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`’. The terms listed in the following tables are declared and axiomatized in this namespace. There are three tables: basic terminology, limit terminology and colimit terminology. As indicated in the left-hand column of these tables, several sub-namespaces are needed. Basic terminology is listed in Table 1. In addition, the special `SET.FTN` term ‘`composable-opspan`’ denotes a particular KIF opspan.

Table 1: Basic terms introduced in the IFF Core Ontology

	Collection	Relation	Function	Example

`SET`	`class`
	`sub-class`
		`subclass`
		`disjoint`
			`binary-union`
			`binary-intersection`
			`power`
				`empty = null`
				`unit = one = terminal`
				`two`
				`three`

`SET .FTN`	`function`
	`endofunction`
	`injection = monomorphism`
	`surjection = epimorphism`
	`bijection = isomorphism`
		`restriction`
		`composable`
		`parallel-pair`
		`isomorphic`
			`source target`
			`fn2rel`
			`counique unique`
			`constant inclusion`
			`class`
			`fiber fiber-inclusion`
			`inverse-image`
			`composition identity`
			`image subfunction`
			`power = direct-image`
			`singleton`
			`left right`
			`union intersection`
			`partition`	

Limit terminology is listed in Table 2.

Table 2: Limit terms introduced in the IFF Core Ontology

	Collection	Relation	Function	Example
SET .LIM	cone		cone-diagram = base vertex component base-shape cone-fiber limiting-cone limit projection mediator tupling-cone tupling terminal = unit unique	
SET .LIM .PRD	diagram = pair cone		class1 class2 opposite cone-diagram vertex first second limiting-cone limit projection1 projection2 mediator pairing tau-cone tau	
SET .LIM .EQU	diagram = parallel-pair cone		source target function1 function2 cone-diagram vertex function limiting-cone limit canon mediator kernel-diagram kernel	
SET .LIM .PBK	diagram = opspan cone	subopspan	class1 class2 opvertex opfirst opsecond pair relation opposite cone-diagram vertex first second limiting-cone limit projection1 projection2 mediator tripling pairing binary-product-opspan tau-cone tau kernel-pair-diagram kernel-pair	
			fiber fiber1 fiber2 fiber12 fiber21 fiber-embedding fiber1-embedding fiber2-embedding fiber12-embedding fiber21-embedding fiber1-projection fiber2-projection	
SET .LIM .SEQU	lax-diagram lax-cone		order source function1 function2 lax parallel-pair lax-cone-diagram vertex function lax-limiting-cone lax-limit subequalizer subcanon mediator	

Colimit terminology is listed in Table 3.

Table 3: Colimits terms introduced in the IFF Core Ontology

	Collection	Relation	Function	Example

SET .COL	`cocone`		`cocone-diagram = base`
			`opvertex`
			`component`
			`base-shape cocone-fiber`
			`colimiting-cocone`
			`colimit injection`
			`comediator`
			`cotupling-cocone cotupling`
			`initial = null counique`	

SET .COL .COPRD	`diagram = pair`
	`cocone`
			`class1 class2 opposite`
			`cocone-diagram opvertex`
			`opfirst opsecond`
			`colimiting-cocone colimit`
			`injection1 injection2`
			`comediator copairing`
			`tau-cone tau`	

SET .COL .COEQ	`diagram = parallel-pair`
	`cocone`
			`source target`
			`function1 function2`
			`cocone-diagram opvertex`
			`function`
			`colimiting-cocone colimit`
			`canon`
			`comediator`	

SET .COL .PSH	`diagram = span`
	`cocone`
			`class1 class2`
			`vertex first second`
			`pair opposite`
			`cocone-diagram opvertex`
			`opfirst opsecond`
			`colimiting-cocone colimit`
			`injection1 injection2`
			`comediator`
			`cotripling copairing`
			`binary-coproduct-opspan`
			`tau-cone tau`	

The signatures for some of the relations and functions in the IFF Core Ontology are listed in Table 4.

Table 4: Signatures for some relations and functions in the conglomerate and core namespaces

Relation	Unary Function	Binary Function
$subclass \subseteq class \times class$ $disjoint \subseteq class \times class$ $restriction \subseteq function \times function$	$source, target : function \rightarrow class$ $identity, range : function \rightarrow class$ $vertex : span \rightarrow class$ $first, second : span \rightarrow function$ $opvertex : opspan \rightarrow class$ $opfirst, opsecond : opspan \rightarrow function$ $opposite : opspan \rightarrow opspan$	$composition : function \times function \rightarrow function$
	$unique : class \rightarrow function$ $cone\text{-}opspan : cone \rightarrow opspan$ $vertex : cone \rightarrow class$ $first, second, mediator : cone \rightarrow function$ $limiting\text{-}cone : opspan \rightarrow cone$	$binary\text{-}product : class \times class \rightarrow class$ $binary\text{-}product\text{-}opspan : class \times class \rightarrow opspan$
	$power : class \rightarrow relation$	

Table 5 (needs much expansion) lists the correspondence between standard mathematical notation and the ontological terminology in the namespace for classes, functions, and finite limits.

Table 5: Correspondence between Mathematical Notation and Ontological Terminology

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

Classes

SET

The collection of all classes is denoted by *class*.

- 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 Set and Classification. Semantically, every class is a set-theoretic collection; hence, syntactically, every class is represented as a KIF collection. The collection of all classes is not a class.

```
(1) (KIF$collection class)
    (KIF$subcollection class KIF$collection)
    (not (class class))
```

- There is an *empty* class. This is an initial class. There is a *unit* class. This is a terminal class. There is a class with two members.

```
(2) (class empty)
    (class null)
    (= empty null)
    (= empty KIF$empty)

(3) (class unit)
    (class one)
    (class terminal)
    (= one unit)
    (= unit terminal)
    (= unit KIF$unit)

(4) (class two)
    (= two KIF$two)

(5) (class three)
    (= three KIF$three)
```

- A *subclass* relation restricts the KIF subcollection relation to classes.

```
(6) (KIF$relation subclass)
    (= (KIF$collection1 subclass) class)
    (= (KIF$collection2 subclass) class)
    (KIF$abridgment subclass KIF$subcollection)
```

- The extent of the subclass is named.

```
(7) (KIF$collection sub-class)
    (KIF$subcollection sub-class KIF$sub-collection)
    (= sub-class (KIF$extent subclass))
```

- A *disjoint* relation restricts the KIF disjoint relation to classes.

```
(3) (KIF$relation disjoint)
    (= (KIF$collection1 disjoint) class)
    (= (KIF$collection2 disjoint) class)
    (KIF$abridgment disjoint KIF$disjoint)
```

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

```
(4) (KIF$function binary-union)
    (= (KIF$source binary-union) (KIF$binary-product [class class]))
    (= (KIF$target binary-union) class)
    (KIF$restriction binary-union KIF$binary-union)

(5) (KIF$function binary-intersection)
    (= (KIF$source binary-intersection) (KIF$binary-product [class class]))
    (= (KIF$target binary-intersection) class)
    (KIF$restriction binary-intersection KIF$binary-intersection)
```

- 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 quasitopos (note, however that we do not use the particular terminology for subobject classifiers, only the instance power terminology in the Classification Ontology). 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. However, for now we only define the fibrational structure that seems to be required. For any class *C* the *power-class* over *C* is the collection of all subclasses of *C*. A *power* function maps a class to its associated power class.

```
(6) (KIF$function power)
    (= (KIF$source power) class)
    (= (KIF$target power) class)
    (forall (?c (class ?c) ?d)
        (<=> ((power ?c) ?d) (subclass ?d ?c)))
```

Functions

SET.FTN

$$X \xrightarrow{f} Y$$

Figure 1: Class Function

A class function (Figure 1) is a special case of a KIF function whose source and target collections are classes. A class function is intended to be an abstract semantic notion. Syntactically however, every class function is represented as a KIF function. The source and target of class functions, considered to be KIF functions, is given by their SET source and target. A class 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 extent of the underlying *relation* of the function. We use the notation $f : X \rightarrow Y$ to indicate the source-target typing of a class function. We use the notation $f(x) = y$ for this instance.

For class functions, both composition and identities are defined. Given two functions $f : X \rightarrow Y$ and $g : Y \rightarrow Z$ the *composition* function $f \cdot g : X \rightarrow Z$ 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 $id_X : X \rightarrow X$. 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 KIF collections.

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

```
(1) (KIF$collection function)
    (KIF$subcollection function KIF$function)

(2) (KIF$function source)
    (= (KIF$source source) function)
    (= (KIF$target source) SET$class)
    (KIF$restriction source KIF$source)

(3) (KIF$function target)
    (= (KIF$source target) function)
    (= (KIF$target target) SET$class)
    (KIF$restriction target KIF$target)
```

- Any function can be embedded as a binary relation.

```
(4) (KIF$function fn2rel)
    (= (KIF$source fn2rel) function)
    (= (KIF$target fn2rel) REL$relation)
    (KIF$restriction fn2rel KIF$fn2rel)
```

In more detail, this restriction can be expressed as follows.

```
    (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 $f_1 : C_1 \rightarrow D_1$ is a *restriction* of a class function $f_2 : C_2 \rightarrow D_2$ when the source (target) of f_1 is a subclass of the source (target) of f_2 and the functions agree (on source elements of f_1); that is, the functions commute (Diagram 1) with the source/target inclusions. Restriction is a constraint on the larger function – it says that the larger function maps the source class of the smaller function into the target class of the smaller function.

$$\begin{tikzcd} C_1 \arrow[r, hook] \arrow[d, "f_1"'] & C_2 \arrow[d, "f_2"] \\ D_1 \arrow[r, hook] & D_2 \end{tikzcd}$$

Diagram 1: Restriction

```
(5) (KIF$relation restriction)
    (= (KIF$collection1 restriction) function)
    (= (KIF$collection2 restriction) function)
    (KIF$abridgment restriction KIF$restriction)
```

In more detail, this abridgment can be expressed as follows.

```
(forall (?f1 (function ?f1) ?f2 (function ?f2))
    (<=> (restriction ?f1 ?f2)
         (and (subclass (source ?f1) (source ?f2))
              (subclass (target ?f1) (target ?f2))
              (forall (?x ((source ?f1) ?x))
                  (= (?f1 ?x) (?f2 ?x))))))
```

○ One can show that one function is a restriction of another function <u>iff</u> the relation associated with the first function is an abridgment of the relation associated with the second functions.

```
(forall (?f1 (function ?f1) ?f2 (function ?f2))
    (<=> (restriction ?f1 ?f2)
         (REL$abridgment (fn2rel ?f1) (fn2rel ?f2))))
```

o For any class *C* there is a *unique* function from *C* to the *unit* class. This is unique.

```
(6) (KIF$function unique)
    (= (KIF$source unique) SET$class)
    (= (KIF$target unique) function)
    (forall (?c (SET$class ?c))
        (and (= (source (counique ?c)) ?c)
             (= (target (counique ?c)) SET$unit)
    (KIF$restriction unique KIF$unique)
```

o For any class *C* there is an *empty* (*counique*) function from the *empty* class to *C*. This is unique.

```
(7) (KIF$function counique)
    (= (KIF$source counique) SET$class)
    (= (KIF$target counique) function)
    (forall (?c (SET$class ?c))
        (and (= (source (counique ?c)) SET$empty)
             (= (target (counique ?c)) ?c)))
    (KIF$restriction counique KIF$counique)
```

o For any two classes *C* and *D* and any element $x \in C$ there is a *constant x* function from *C* to *D*.

```
(8) (KIF$function constant)
    (= (KIF$source constant) (SET.LIM.PRD$binary-product SET$class)
    (= (KIF$target constant) KIF$function)
    (forall (?c (SET$class ?c) ?d (SET$class ?d))
        (and (= (KIF$source (constant [?c ?d])) ?c)
             (= (KIF$target (constant [?c ?d])) function)
             (forall (?x (?c ?x))
                 (and (= (source ((constant [?c ?d]) ?x)) ?c)
                      (= (target ((constant [?c ?d]) ?x)) ?d)
                      (forall (?y (?c ?y))
                          (= (((constant [?c ?d]) ?x) ?y) ?x))))))
    (KIF$restriction constant KIF$constant)
```

o For any two classes that are ordered by inclusion $A \subseteq B$ there is an *inclusion* function $A \rightarrow B$.

```
(9) (KIF$function inclusion)
    (= (KIF$source inclusion) sub-class)
    (= (KIF$target inclusion) function)
    (forall (?a ?b (sub-class [?a b]))
        (and (= (source (inclusion [?a ?b])) ?a)
             (= (target (inclusion [?a ?b])) ?b)))
    (KIF$restriction inclusion KIF$inclusion)
```

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

```
(6) (KIF$collection endofunction)
    (KIF$subcollection endofunction function)

(7) (KIF$function class)
    (= (KIF$source class) endofunction)
    (= (KIF$target class) SET$class)
    (forall (?f (endofunction ?f))
        (and (KIF$restriction class source)
             (KIF$restriction class target)))
```

- For any class function $f : A \rightarrow B$, 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 $\subseteq_{f,y} : f^{-1}(y) \rightarrow A$ for any element $y \in B$. We do not state that the class fiber function is a restriction of the KIF fiber function, since we do not assume the existence of power collections.

```
(9) (KIF$function fiber)
    (KIF$source fiber function)
    (KIF$target fiber function)
    (forall (?f (function ?f))
        (and (= (source (fiber ?f)) (target ?f))
             (= (target (fiber ?f)) (SET$power (source ?f)))))
    (forall (?f (function ?f)
             ?y ((target ?f) ?y)
             ?x ((source ?f) ?x))
        (<=> (((fiber ?f) ?y) ?x)
             (= (?f ?x) ?y))))

(10) (KIF$function fiber-inclusion)
     (KIF$source fiber-inclusion function)
     (KIF$target fiber-inclusion function KIF$function)
     (forall (?f (function ?f))
         (and (= (KIF$source (fiber-inclusion ?f) (target ?f))
              (= (KIF$target (fiber-inclusion ?f) function)
              (forall (?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)]))))))
```

- 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. This takes two forms: the direct image and the inverse image. For any class function $f : A \rightarrow B$ there is an *inverse image* function $f^{-1} : \wp B \rightarrow \wp A$ defined by $f^{-1}(Y) = \{x \in A \mid f(x) \in Y\} \subseteq A$ for any subset $Y \subseteq B$.

```
(11) (KIF$function inverse-image)
     (= (KIF$source inverse-image) function)
     (= (KIF$target inverse-image) function)
     (forall (?f (function ?f))
         (and (= (source (inverse-image ?f)) (SET$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))))
```

- Two class functions are *composable* when the target of the first is equal to the source of the second. The *composition* of two composable functions $f_1 : A \rightarrow B$ and $f_2 : B \rightarrow C$ is the class function $f_1 \cdot f_2 : A \rightarrow C$ defined by $f_1 \cdot f_2\,(x) = f_2(f_1(x))$ for any element $x \in A$.

```
(12) (KIF$opspan composable-opspan)
     (= composable-opspan [target source])

(13) (KIF$relation composable)
     (= (KIF$collection1 composable) function)
     (= (KIF$collection2 composable) function)
     (= (KIF$extent composable) (KIF$pullback composable-opspan))

(14) (KIF$function composition)
     (= (KIF$source composition) (KIF$pullback composable-opspan))
     (= (KIF$target composition) function)
     (forall (?f1 (function ?f1) ?f2 (function ?f2) (composable ?f1 ?f2))
         (and (= (source (composition [?f1 ?f2])) (source ?f1))
              (= (target (composition [?f1 ?f2])) (target ?f2))
              (forall (?x ((source ?f1) ?x))
                  (= ((composition [?f1 ?f2]) ?x) (?f2 (?f1 ?x))))))
```

- Composition satisfies the usual *associative law*.

```
(forall (?f1 (function ?f1) ?f2 (function ?f2) ?f3 (function ?f3)
         (composable ?f1 ?f2) (composable ?f2 ?f3))
    (= (composition [?f1 (composition [?f2 ?f3])])
       (composition [(composition [?f1 ?f2]) ?f3])))
```

- For any class *C* there is an *identity* class function.

```
(15) (KIF$function identity)
     (= (KIF$source identity) SET$class)
     (= (KIF$target identity) function)
     (forall (?c (SET$class ?c))
         (and (= (source (identity ?c)) ?c)
              (= (target (identity ?c)) ?c)
              (forall (?x (?c ?x))
                  (= ((identity ?c) ?x) ?x))))
```

- 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.

```
(16) (KIF$relation parallel-pair)
     (= (KIF$collection1 parallel-pair) function)
     (= (KIF$collection2 parallel-pair) function)
     (forall (?f (function ?f) ?g (function ?g))
         (<=> (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.

```
(17) (KIF$collection injection)
     (= injection (KIF$binary-intersection [function KIF$injection]))

(18) (KIF$collection monomorphism)
     (KIF$subcollection monomorphism function)
     (forall (?f (function ?f))
         (<=> (monomorphism ?f)
              (forall (?g1 (function ?g1) ?g2 (function ?g2))
                  (=> (and (composable ?g1 ?f) (composable ?g2 ?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.

```
(19) (KIF$collection surjection)
     (= surjection (KIF$binary-intersection [function KIF$surjection]))

(20) (KIF$collection epimorphism)
     (KIF$subcollection epimorphism function)
     (forall (?f (function ?f))
         (<=> (epimorphism ?f)
              (forall (?g1 (function ?g1)
                       ?g2 (function ?g2))
                  (=> (and (composable ?f ?g1)
                           (composable ?f ?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.

```
(21) (KIF$collection bijection)
     (= bijection (KIF$binary-intersection [injection surjection]))

(22) (KIF$collection isomorphism)
     (= isomorphism (KIF$binary-intersection [monomorphism epimorphism]))
```

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

```
    (= bijection isomorphism)
```

- Two classes are isomorphic when there is an isomorphism between them.

```
(23) (KIF$relation isomorphic)
     (= (KIF$collection1 isomorphic) SET$class)
     (= (KIF$collection2 isomorphic) SET$class)
     (KIF$abridgment isomorphic KIF$isomorphic)
```

- The image class of the function $f : A \rightarrow B$ is the class $f[A] = \{y \in B \mid \exists x \in A, y = f(x)\} \subseteq B$.

```
(24) (KIF$function image)
     (= (KIF$source image) function)
     (= (KIF$target image) SET$class)
     (forall (?f (function ?f))
         (and (SET$subclass (image ?f) (target ?f))
              (forall (?y ((target ?f) ?y))
                  (<=> ((image ?f) ?y)
                       (exists (?x ((source ?f) ?x))
                           (= ?y (?f ?x)))))))
```

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

```
(25) (KIF$function subfunction)
     (= (KIF$source subfunction) ORD$preorder)
     (= (KIF$target subfunction) KIF$relation)
     (forall (?o (ORD$preorder ?o))
         (and (= (KIF$collection1 (subfunction ?o)) function)
              (= (KIF$collection2 (subfunction ?o)) function)
              (forall (?f1 (function ?f2)
                       ?f2 (function ?f2))
                  (<=> ((subfunction ?o) ?f1 ?f2)
                       (and (= (source ?f1) (source ?f2))
                            (= (target ?f1) (target ?f2))
                            (= (target ?f1) (ORD$class ?o))
                            (forall (?x ((source ?f1) ?x))
                                (?o (?f1 ?x) (?f2 ?x))))))))
```

- For any class function $f : A \rightarrow B$ the *direct image* function $\wp f : \wp A \rightarrow \wp B$ is defined by $\wp f(X) = \{y \in B \mid y = f(x) \text{ some } x \in X\} \subseteq B$ for any subset $X \subseteq A$.

```
(26) (KIF$function power)
     (KIF$function direct-image)
     (= power direct-image)
     (= (KIF$source power) function)
     (= (KIF$target power) 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, image is related to power as follows.

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

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

```
(27) (KIF$function singleton)
     (= (KIF$source singleton) SET$class)
     (= (KIF$target singleton) function)
     (forall (?c (SET$class ?c))
         (and (= (source (singleton ?c)) ?c)
              (= (target (singleton ?c)) (SET$power ?c))
              (forall (?x (?c ?x))
                  (= ((singleton ?c) ?x) ?x))))
```

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

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

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

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

```
(28) (KIF$function left)
     (= (KIF$source left) (KIF$pullback [target ORD$class]))
     (= (KIF$target left) REL$relation)
     (forall (?f (function ?f) ?o (ORD$preorder ?o) (= (target ?f) (ORD$class ?o)))
         (and (= (REL$source (left [?f ?o])) (target ?f))
              (= (REL$target (left [?f ?o])) (source ?f))
              (forall (?a ((target ?f) ?a)
                       ?b ((source ?f) ?b))
                  (<=> ((left [?f ?o]) ?a ?b)
                       (?o ?a (?f ?b))))))

(29) (KIF$function right)
     (= (KIF$source right) (KIF$pullback [target ORD$class]))
     (= (KIF$target right) REL$relation)
     (forall (?f (function ?f) ?o (ORD$preorder ?o) (= (target ?f) (ORD$class ?o)))
         (and (= (REL$source (right [?f ?o])) (source ?f))
              (= (REL$target (right [?f ?o])) (target ?f))
              (forall (?b ((source ?f) ?b)
                       ?a ((target ?f) ?a))
                  (<=> ((right [?f ?o]) ?b ?a)
                       (?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 [?f (ORD$identity (target ?f))])))

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

- For any class C, there is a *union* operator $\cup_C : \wp\wp C \rightarrow \wp C$ and an *intersection* operator $\cap_C : \wp\wp C \rightarrow \wp C$. 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 $\cap_C(S)$.

```
(30) (KIF$function union)
     (= (KIF$source union) SET$class)
     (= (KIF$target union) 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))))))

(31) (KIF$function intersection)
```

```
        (= (KIF$source intersection) SET$class)
        (= (KIF$target intersection) function)
        (forall (?c (SET$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))))))
```

- Any class can be partitioned. A *partition* function maps a class C to its collection of partitions $P \in \wp\wp C$.

```
(32) (KIF$function partition)
     (= (KIF$source partition) class)
     (= (KIF$target partition) class)
     (forall (?c (SET$class ?c))
         (subclass (partition ?c) (SET$power (SET$power ?c))))
     (forall (?c (class ?c) ?p ((SET$power (SET$power ?c)) ?p))
         (<=> ((partition ?c) ?p)
              (and (= ((union ?c) ?p) ?c)
                   (forall (?pj (?p ?pj) ?pk (?p ?pk) (not (= ?pj ?pk)))
                       (SET$disjoint ?pj ?pk)))))
```

Limits

SET.LIM

Here we present axioms that make the quasicategory of classes and functions 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. This commonality has been abstracted into a general formulation of limits. The *diagrams* and *limits* are denoted by both generic and specific terminology. A *limit* is the vertex of a limiting diagram of a certain shape. The base diagram for a limit is represented by the diagram terminology in the (large) graph namespace in the IFF Category Theory Ontology.

$$\begin{tikzcd}
\mathit{cone}(G_k) \arrow[r] \arrow[d, shift left, "\mathit{base}_k"] & \mathit{cone} \arrow[d, shift left, "\mathit{base}"] \\
\mathit{diagram}(G_k) \arrow[r] \arrow[u, shift left, "\mathit{limit}_k"] & \mathit{diagram} \arrow[u, shift left, "\mathit{limit}"] \arrow[d, "\mathit{shape}"] \\
& \mathit{graph}
\end{tikzcd}$$

Diagram 2: Diagrams, Cones, and Fibers

In various places below, we use definite descriptions. Here we paraphrase Chris Menzel. Definite descriptions are not an official part of the KIF language, since adding them requires modifying the semantics of KIF to allow for non-denoting terms (as many descriptions do not denote anything). Hence, they are better regarded as convenient abbreviations that can be unpacked a la Russell's theory of descriptions. Let '`s1`', ..., '`sn`' and '`s'`' be sentences, typically containing free occurrences of variable '`v`'. Then

```
(p t1 ... (the (v s1, ..., sn) s') ... tm)
```

is an abbreviation for

```
(exists (v')
    (and (forall (v)
             (<=> (and s1 ... sn s')
                  (= v v')))
         (p t1 ... v' ... tm)))
```

$$\begin{tikzcd}
& D(n) \arrow[dd, "D(e)"] & & n \arrow[dd, "e"] \\
C \arrow[ur, "\tau(n)"] \arrow[dr, "\tau(n')"'] & & & \\
& D(n') & & n'
\end{tikzcd}$$

Diagram 3: Cone

- A *cone* (Diagram 3) consists of a base *diagram*, a *vertex*, and a collection of *component* functions indexed by the nodes in the shape of the diagram. The cone is situated over the base diagram. The component functions form commutative diagrams with the diagram functions.

```
(1) (KIF$collection cone)

(2) (KIF$function cone-diagram)
    (KIF$function base)
    (= cone-diagram base)
    (= (KIF$source cone-diagram) cone)
    (= (KIF$target cone-diagram) GPH$diagram)

(3) (KIF$function vertex)
    (= (KIF$source vertex) cone)
    (= (KIF$target vertex) SET$class)

(4) (KIF$function component)
    (= (KIF$source component) cone)
    (= (KIF$target component) KIF$function)
    (forall (?r (cone ?r))
        (and (= (KIF$source (component ?r)) (GPH$node (GPH$shape (cone-diagram ?r))))
             (= (KIF$target (component ?r)) SET.FTN$function)
             (forall (?n ((GPH$node (GPH$shape (cone-diagram ?r))) ?n))
                 (and (= (SET.FTN$source ((component ?r) ?n)) (vertex ?r))
                      (= (SET.FTN$target ((component ?r) ?n))
                         ((GPH$class (cone-diagram ?r)) ?n))))
             (forall (?e ((GPH$edge (GPH$shape (cone-diagram ?r))) ?e))
                 (= (SET.FTN$composition
                        ((component ?r) ((GPH$source (GPH$shape (cone-diagram ?r))) ?e))
                        ((GPH$function (cone-diagram ?r)) ?e))
                    ((component ?r) ((GPH$target (GPH$shape (cone-diagram ?r))) ?e))))))
```

- The *cone-fiber* cone(G) of any graph G is the collection of all cones whose base diagram has shape G.

```
(5) (KIF$function base-shape)
    (= (KIF$source base-shape) cone)
    (= (KIF$target base-shape) GPH$graph)
    (forall (?r (cone ?r))
        (= (base-shape ?r) (GPH$shape (base ?r))))

(6) (KIF$function cone-fiber)
    (= (KIF$source cone-fiber) GPH$graph))
    (= (KIF$target cone-fiber) KIF$collection)
    (= cone-fiber (KIF$fiber base-shape))
```

- The KIF function '`limiting-cone`' maps a diagram to its limit (limiting cone) (Diagram 4). This asserts that a limit exists for any diagram. The universality of this limit is expressed by axioms for the mediator function. The vertex of the limiting cone is a specific *limit* class given by the KIF function '`limit`'. It comes equipped with component projection functions. This notation is for convenience of reference. Axiom (#) ensures that this limit is specific, the Cartesian product. Axiom (%) ensures that the component projection functions are also specific, the projections from the Cartesian product.

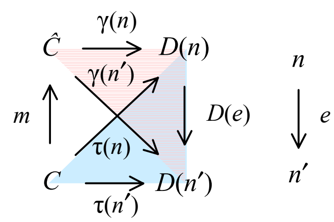

Diagram 4: Limiting Cone

```
(7) (KIF$function limiting-cone)
    (= (KIF$source limiting-cone) GPH$diagram)
    (= (KIF$target limiting-cone) cone)
    (forall (?d (GPH$diagram ?d))
        (= (cone-diagram (limiting-cone ?d)) ?d))

(8) (KIF$function limit)
    (= (KIF$source limit) GPH$diagram)
    (= (KIF$target limit) SET$class)
    (forall (?d (GPH$diagram ?d))
        (= (limit ?d) (vertex (limiting-cone ?d))))
(#) (forall (?d (GPH$diagram ?d))
         (SET$subclass (limit ?d) (KIF$product (GPH$class ?d))))

(9) (KIF$function projection)
    (= (KIF$source projection) GPH$diagram)
    (= (KIF$target projection) KIF$function)
    (forall (?d (GPH$diagram ?d))
        (and (= (KIF$source (projection ?d)) (GPH$node (GPH$shape ?d)))
             (= (KIF$target (projection ?d)) SET.FTN$function)
             (forall (?n ((GPH$node (GPH$shape ?d)) ?n))
                 (and (= (SET.FTN$source ((projection ?d) ?n)) (limit ?d))
                      (= (SET.FTN$target ((projection ?d) ?n)) ((GPH$class ?d) ?n))
                      (= ((projection ?d) ?n)
                         ((component (limiting-cone ?d)) ?n))))))
(%) (forall (?d (GPH$diagram ?d)
             ?t ((limit ?d) ?t)
             ?n ((GPH$node (GPH$shape ?d)) ?n))
        (= (((projection ?d) ?n) ?t) (?t ?n)))
```

- There is a *mediator* function from the vertex of a cone over a diagram to the limit of the diagram. This is the unique function that commutes with the component functions of the cone. We use a KIF definite description to define this. Existence and uniqueness represents the universality of the limit operator. We have also introduced a <u>convenience term</u> '`tupling`'. With a diagram parameter, the KIF function '`(tupling ?d)`' maps a tuple of class functions, that form a cone over the diagram, to their mediator (tupling) function.

```
(10) (KIF$function mediator)
     (= (KIF$source mediator) cone)
     (= (KIF$target mediator) SET.FTN$function)
     (forall (?r (cone ?r))
        (= (mediator ?r)
           (the (?f (SET.FTN$function ?f))
                (and (= (SET.FTN$source ?f) (vertex ?r))
```

```
                        (= (SET.FTN$target ?f) (limit (cone-diagram ?r)))
                        (forall (?n ((GPH$node (GPH$shape (cone-diagram ?r))) ?n))
                            (= (SET.FTN$composition
                                   ?f ((projection (cone-diagram ?r)) ?n))
                               ((component ?r) ?n)))))))

(11) (KIF$function tupling-cone)
     (KIF$source tupling-cone) GPH$diagram)
     (KIF$target tupling-cone) KIF$partial-function)
     (forall (?d (GPH$diagram ?d))
         (and (= (KIF$source (tupling-cone ?d))
                 (KIF$power [(GPH$node (GPH$shape ?d)) SET.FTN$function]))
              (= (KIF$target (tupling-cone ?d)) cone)
              (forall (?f ((KIF$power [(GPH$node (GPH$shape ?d)) SET.FTN$function]) ?f))
                  (<=> ((KIF$domain (tupling-cone ?d)) ?f)
                       (and (exists (?c (collection ?c))
                                (forall (?n ((GPH$node (GPH$shape ?d)) ?n))
                                    (= (SET.FTN$source (?f ?n)) ?c)))
                            (forall (?n ((GPH$node (GPH$shape ?d)) ?n))
                                (= (SET.FTN$target (?f ?n)) ((GPH$class ?d) ?n)))
                            (forall (?e ((GPH$edge (GPH$shape ?d)) ?e))
                                (= (SET.FTN$composition
                                       (?f ((GPH$source (GPH$shape ?d)) ?e))
                                       ((GPH$function ?d) ?e))
                                   (?f ((GPH$target (GPH$shape ?d)) ?e)))))))))
     (forall (?d (GPH$diagram ?d)
              ?f ((KIF$domain (tupling-cone ?d)) ?f))
         (and (= (cone-diagram ((tupling-cone ?d) ?f)) ?d)
              (forall (?n ((GPH$node (GPH$shape ?d)) ?n))
                  (and (= (vertex ((tupling-cone ?d) ?f)) (SET.FTN$source (?f ?n)))
                       (= ((component ((tupling-cone ?d) ?f)) ?n) (?f ?n))))))

(12) (KIF$function tupling)
     (= (KIF$source tupling) GPH$diagram)
     (= (KIF$target tupling) KIF$partial-function)
     (forall (?d (GPH$diagram ?d))
         (and (= (KIF$source (tupling ?d))
                 (KIF$power [(GPH$node (GPH$shape ?d)) SET.FTN$function]))
              (= (KIF$target (tupling ?d)) SET.FTN$function)
              (= (KIF$domain (tupling ?d)) (KIF$domain (tupling-cone ?d)))
              (forall ?f ((KIF$domain (tupling ?d)) ?f))
                  (= ((tupling ?d) ?f)
                     (mediator ((tupling-cone ?d) ?f))))))
```

The Terminal Class

- A cone, whose base diagram is the diagram of empty shape, is essentially just a class – the vertex class of the cone. There is an isomorphism between cones over the empty diagram and classes.

```
(SET.FTN$isomorphic (SET.LIM$cone-fiber GPH$empty) SET$class)
```

- The limit (there is only one, since there is only one diagram) is special.

```
(13) (SET$class terminal)
     (SET$class unit)
     (= terminal unit)
     (= terminal (limit GPH$empty-diagram))
```

- The mediator of any class (cone) is the unique function from that class to the terminal class. Therefore, the limit is the unit or terminal class. For each class C there is a *unique* function $!_C : C \rightarrow 1$ to the unit class.

```
(14) (KIF$function unique)
     (= (KIF$source unique) SET$class)
     (= (KIF$target unique) SET.FTN$function)
     (forall (?c (SET$class ?c))
         (= (unique ?c)
            (the (?f (SET.FTN$function ?f))
                 (and (= (SET.FTN$source ?f) ?c)
                      (= (SET.FTN$target ?f) unit)))))
```

- The following facts can be proven: the limit is the terminal class, and the mediator is the unique function.

```
(= terminal SET$terminal)
(= unique SET.FTN$unique)
```

Binary Products

SET.LIM.PRD

A *binary product* (Figure 2) is a finite limit for a diagram of shape *two* = • •. Such a diagram (of classes and functions) is called a *pair* of classes.

$$\begin{tikzcd} & C_1 \times C_2 \arrow[dl, "\pi_1"'] \arrow[dr, "\pi_2"] & \\ C_1 & & C_2 \end{tikzcd}$$

Figure 2: Binary Product

- 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*. We use either the generic term '`diagram`' or the specific term '`pair`' to denote the *pair* collection. A pair is the special case of a general diagram of shape *two*.

```
(1) (KIF$collection diagram)
    (KIF$collection pair)
    (= pair diagram)
    (KIF$subcollection diagram GPH$diagram)
    (= diagram (GPH$diagram-fiber GPH$two))

(2) (KIF$function class1)
    (= (KIF$source class1) diagram)
    (= (KIF$target class1) SET$class)
    (forall (?d (diagram ?d))
        (= (class1 ?d) ((GPH$class ?d) 1)))

(3) (KIF$function class2)
    (= (KIF$source class2) diagram)
    (= (KIF$target class2) SET$class)
    (forall (?d (diagram ?d))
        (= (class2 ?d) ((GPH$class ?d) 2)))
```

- By the unique determination inherited from the general case, we can prove the isomorphism *pair* ≅ *class* ×*class*. This *pair* notion is abstract, and hence is not a subcollection of the KIF *pair* collection.

```
(KIF$isomorphic pair (KIF$binary-product [SET$class SET$class]))
```

- Every pair has an opposite.

```
(4) (KIF$function opposite)
    (= (KIF$source opposite) pair)
    (= (KIF$target opposite) 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 *binary product cone* consists of a pair of functions called *first* and *second*. These are required to have a common source class called the *vertex* of the cone. Each binary product cone is situated over a binary product diagram (pair). A binary product cone is the special case of a general cone over a binary product diagram (pair of classes).

```
(5) (KIF$collection cone)
    (KIF$subcollection cone SET.LIM$cone)
    (= cone (SET.LIM$cone-fiber GPH$two))

(6) (KIF$function cone-diagram)
    (= (KIF$source cone-diagram) cone)
    (= (KIF$target cone-diagram) diagram)
    (SET.FTN$restriction cone-diagram SET.LIM$cone-diagram)

(7) (KIF$function vertex)
    (= (KIF$source vertex) cone)
```

```
    (= (KIF$target vertex) SET$class)
    (SET.FTN$restriction vertex SET.LIM$vertex)

(8) (KIF$function first)
    (= (KIF$source first) cone)
    (= (KIF$target first) SET.FTN$function)
    (forall (?r (cone ?r))
        (= (first ?r) ((SET.LIM$component ?r) 1)))

(9) (KIF$function second)
    (= (KIF$source second) cone)
    (= (KIF$target second) SET.FTN$function)
    (forall (?r (cone ?r))
        (= (second ?r) ((SET.LIM$component ?r) 2)))
```

- The KIF function ‘`limiting-cone`’ maps a pair of classes to its binary product (limiting binary product cone) (Figure 2). A limiting binary product cone is the special case of a general limiting cone over a binary product diagram (pair of classes).

```
(10) (KIF$function limiting-cone)
     (= (KIF$source limiting-cone) diagram)
     (= (KIF$target limiting-cone) cone)
     (KIF$restriction limiting-cone SET.LIM$limiting-cone)

(11) (KIF$function limit)
     (KIF$function binary-product)
     (= binary-product limit)
     (= (KIF$source limit) diagram)
     (= (KIF$target limit) SET$class)
     (forall (?d (diagram ?d))
         (= (limit ?d) (vertex (limiting-cone ?d))))

(12) (KIF$function projection1)
     (= (KIF$source projection1) diagram)
     (= (KIF$target projection1) SET.FTN$function)
     (forall (?d (GPH$diagram ?d)
         (= (projection1 ?d) (first (limiting-cone ?d))))

(13) (KIF$function projection2)
     (= (KIF$source projection2) diagram)
     (= (KIF$target projection2) SET.FTN$function)
     (forall (?d (GPH$diagram ?d)
         (= (projection2 ?d) (second (limiting-cone ?d))))
```

- There is a *mediator* function from the vertex of a binary product cone over a binary product diagram (pair of classes) to the binary product of the pair. This is the unique function that commutes with the component functions of the cone. We have also introduced a “convenience term” <u>pairing</u>. With a diagram parameter, this maps a pair of class functions, which form a binary product cone with the diagram, to their mediator (or *pairing*) function.

```
(14) (KIF$function mediator)
     (= (KIF$source mediator) cone)
     (= (KIF$target mediator) SET.FTN$function)
     (KIF$restriction mediator SET.LIM$mediator)

(15) (KIF$function pairing)
     (= (KIF$source pairing) diagram)
     (= (KIF$target pairing) KIF$partial-function)
     (forall (?d (diagram ?d))
         (and (= (KIF$source (pairing ?d))
                 (KIF$power [two SET.FTN$function]))
              (= (KIF$target (pairing ?d)) SET.FTN$function)
              (forall (?f1 ?f2 ((KIF$power [two SET.FTN$function]) [?f1 ?f2]))
                  (<=> ((KIF$domain (pairing ?d)) [?f1 ?f2])
                       (and (SET.FTN$function ?f1)
                            (SET.FTN$function ?f2)
                            (= (SET.FTN$source ?f1) (SET.FTN$source ?f2))
                            (= (SET.FTN$target ?f1) (class1 ?d))
                            (= (SET.FTN$target ?f2) (class2 ?d)))))))
     (KIF$restriction pairing SET.LIM$tupling)
```

- 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 (for products).

```
(16) (KIF$function tau-cone)
     (= (KIF$source tau-cone) pair)
     (= (KIF$target tau-cone) cone)
     (forall (?p (pair ?p))
        (and (= (cone-diagram (tau-cone ?p)) ?p)
             (= (vertex (tau-cone ?p)) (binary-product (opposite ?p)))
             (= (first (tau-cone ?p)) (projection2 (opposite ?p)))
             (= (second (tau-cone ?p)) (projection1 (opposite ?p)))))

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

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

```
     (forall (?p (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)))))
```

Equalizers

SET.LIM.EQU

An *equalizer* (Figure 3) is a finite limit for a diagram of shape *parallel-pair* = $\cdot \rightrightarrows \cdot$. Such a diagram (of classes and functions) is called a *parallel pair* of functions.

$$\begin{tikzcd} E \arrow[r, "\varepsilon"] \arrow[dr] & C_1 \arrow[d, shift right, "f_1"'] \arrow[d, shift left, "f_2"] \\ & C_2 \end{tikzcd}$$

Figure 3: Equalizer

- 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. We use either the generic term '`diagram`' or the specific term '`parallel-pair`' to denote the *parallel pair* collection. A parallel pair is a special case of a general diagram of shape *parallel-pair*.

```
(1) (KIF$collection diagram)
    (KIF$collection parallel-pair)
    (= parallel-pair diagram)
    (KIF$subcollection diagram GPH$diagram)
    (= diagram (GPH$diagram-fiber GPH$parallel-pair))

(2) (KIF$function source)
    (= (KIF$source source) diagram)
    (= (KIF$target source) SET$class)
    (forall (?d (diagram ?d))
        (= (source ?d) ((GPH$class ?d) 1)))

(3) (KIF$function target)
    (= (KIF$source target) diagram)
    (= (KIF$target target) SET$class)
    (forall (?d (diagram ?d))
        (= (target ?d) ((GPH$class ?d) 2)))

(4) (KIF$function function1)
    (= (KIF$source function1) diagram)
    (= (KIF$target function1) SET.FTN$function)
    (forall (?d (diagram ?d))
        (= (function1 ?d) ((GPH$function ?d) 1)))

(5) (KIF$function function2)
    (= (KIF$source function2) diagram)
    (= (KIF$target function2) SET.FTN$function)
    (forall (?d (diagram ?d))
        (= (function2 ?d) ((GPH$function ?d) 2)))
```

- An *equalizer cone* consists of 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. Each equalizer cone is situated over an equalizer diagram (parallel pair of functions). An equalizer cone is the special case of a general cone over an equalizer diagram.

```
(6) (KIF$collection cone)
    (KIF$subcollection cone SET.LIM$cone)
    (= cone (SET.LIM$ cone-fiber GPH$parallel-pair))

(7) (KIF$function cone-diagram)
    (= (KIF$source cone-diagram) cone)
    (= (KIF$target cone-diagram) diagram)
    (SET.FTN$restriction cone-diagram SET.LIM$cone-diagram)

(8) (KIF$function vertex)
    (= (KIF$source vertex) cone)
    (= (KIF$target vertex) SET$class)
    (SET.FTN$restriction vertex SET.LIM$vertex)

(9) (KIF$function function)
    (= (KIF$source function) cone)
    (= (KIF$target function) SET.FTN$function)
    (forall (?r (cone ?r))
        (= (function ?r) ((SET.LIM$component ?r) 1)))
```

- The KIF function '`limiting-cone`' maps a parallel pair of functions to its equalizer (limiting equalizer cone) (Figure 3). A limiting equalizer cone is the special case of a general limiting cone over an equalizer diagram (parallel pair of functions).

```
(10) (KIF$function limiting-cone)
     (= (KIF$source limiting-cone) diagram)
     (= (KIF$target limiting-cone) cone)
     (KIF$restriction limiting-cone SET.LIM$limiting-cone)

(11) (KIF$function limit)
     (KIF$function equalizer)
     (= equalizer limit)
     (= (KIF$source limit) diagram)
     (= (KIF$target limit) SET$class)
     (forall (?d (diagram ?d))
         (= (limit ?d)
            (vertex (limiting-cone ?d))))

(12) (KIF$function canon)
     (= (KIF$source canon) diagram)
     (= (KIF$target canon) SET.FTN$function)
     (forall (?d (diagram ?d))
         (= (canon ?d) (function (limiting-cone ?p))))
```

- There is a *mediator* function from the vertex of an equalizer cone over an equalizer diagram (parallel pair of functions) to the equalizer of the parallel pair. This is the unique function that commutes with the component functions of the cone.

```
(13) (KIF$function mediator)
     (= (KIF$source mediator) cone)
     (= (KIF$target mediator) SET.FTN$function)
     (KIF$restriction mediator SET.LIM$mediator)
```

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

```
(14) (KIF$function kernel-diagram)
     (= (KIF$source kernel-diagram) SET.FTN$function)
     (= (KIF$target kernel-diagram) 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) (KIF$function kernel)
```

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

Pullbacks

SET.LIM.PBK

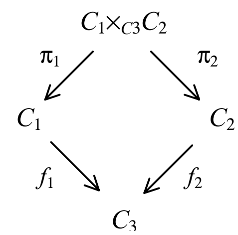
Figure 4: Pullback

A *pullback* (Figure 4) is a finite limit for a diagram of shape *opspan* = • → • ← •. Such a diagram (of classes and functions) is called an *opspan*.

- An *opspan* is the appropriate base diagram for a pullback. 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*. We use either the generic term '`diagram`' or the specific term '`opspan`' to denote the *opspan* collection. An opspan is the special case of a general diagram whose shape is the graph that is also named *opspan*.

```
(1) (KIF$collection diagram)
    (KIF$collection opspan)
    (= opspan diagram)
    (KIF$subcollection diagram GPH$diagram)
    (= diagram (GPH$diagram-fiber GPH$opspan))

(2) (KIF$function class1)
    (= (KIF$source class1) diagram)
    (= (KIF$target class1) SET$class)
    (forall (?d (diagram ?d))
        (= (class1 ?d) ((GPH$class ?d) 1)))

(3) (KIF$function class2)
    (= (KIF$source class2) diagram)
    (= (KIF$target class2) SET$class)
    (forall (?d (diagram ?d))
        (= (class2 ?d) ((GPH$class ?d) 2)))

(4) (KIF$function opvertex)
    (= (KIF$source opvertex) diagram)
    (= (KIF$target opvertex) SET$class)
    (forall (?d (diagram ?d))
        (= (opvertex ?d) ((GPH$class ?d) 3)))

(5) (KIF$function opfirst)
    (= (KIF$source opfirst) diagram)
    (= (KIF$target opfirst) SET.FTN$function)
    (forall (?d (diagram ?d))
        (= (opfirst ?d) ((GPH$function ?d) 1)))

(6) (KIF$function opsecond)
    (= (KIF$source opsecond) diagram)
    (= (KIF$target opsecond) SET.FTN$function)
    (forall (?d (diagram ?d))
        (= (opsecond ?d) ((GPH$function ?d) 2)))
```

- An opspan S_1 is a *subopspan* of an opspan S_2 when the first, second and opvertex component classes of S_1 are subclasses of the corresponding component classes of S_2 and the opfirst and opsecond functions of S_1 are restrictions of the corresponding component functions of S_2.

```
(7) (KIF$relation subopspan)
    (= (collection1 subopspan) function)
    (= (collection2 subopspan) function)
    (forall (?s1 (opspan ?s1) ?s2 (opspan ?s2))
        (<=> (subopspan ?s1 ?s2)
             (and (SET$subclass (class1 ?s1) (class1 ?s2))
                  (SET$subclass (class2 ?s1) (class2 ?s2))
                  (SET$subclass (extent ?s1) (extent ?s2))
                  (SET.FTN$restriction (opfirst ?s1) (opfirst ?s2))
                  (SET.FTN$restriction (opsecond ?s1) (opsecond ?s2)))))
```

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

```
(8) (KIF$function pair)
    (= (KIF$source pair) diagram)
    (= (KIF$target pair) SET.LIM.PRD$diagram)
    (= pair
       (GPH$restriction [GPH$two GPH$opspan]))
```

- Associated with any pullback diagram (opspan) $S = (f_1 : A_1 \rightarrow B,\ f_2 : A_2 \rightarrow B)$ is a relation $rel(S) \subseteq A_1 \times A_2$, whose extent is defined to be the pullback class $\{(x_1, x_2) \mid f_1(x_1) = f_2(x_2)\}$. A *relation* function maps an opspan to its associated relation.

```
(9) (KIF$function relation)
    (= (KIF$source relation) diagram)
    (= (KIF$target relation) REL$relation)
    (forall (?s (opspan ?s))
        (and (= (REL$class1 (relation ?s)) (class1 ?s))
             (= (REL$class2 (relation ?s)) (class2 ?s))
             (forall (?x1 ((class1 ?s) ?x1) ?x2 ((class2 ?s) ?x1))
                 (<=> ((relation ?s) ?x1 ?x2)
                      (= ((opfirst ?s) ?x1) ((opsecond ?s) ?x2))))))
```

- Every opspan has an opposite.

```
(10) (KIF$function opposite)
     (= (KIF$source opposite) opspan)
     (= (KIF$target opposite) 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))
```

- A *pullback cone* consists of an underlying pullback diagram (*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. Each pullback cone is situated over its underlying pullback diagram (opspan). A pullback cone is the special case of a general cone over a pullback diagram (opspan).

```
(11) (KIF$collection cone)
     (KIF$subcollection cone SET.LIM$cone)
     (= cone (SET.LIM$cone-fiber GPH$opspan))

(12) (KIF$function cone-diagram)
     (= (KIF$source cone-diagram) cone)
     (= (KIF$target cone-diagram) diagram)
     (SET.FTN$restriction cone-diagram SET.LIM$cone-diagram)

(13) (KIF$function vertex)
     (= (KIF$source vertex) cone)
     (= (KIF$target vertex) SET$class)
     (SET.FTN$restriction vertex SET.LIM$vertex)

(14) (KIF$function first)
     (= (KIF$source first) cone)
     (= (KIF$target first) SET.FTN$function)
     (forall (?r (cone ?r))
         (= (first ?r) ((SET.LIM$component ?r) 1)))

(15) (KIF$function second)
     (= (KIF$source second) cone)
     (= (KIF$target second) SET.FTN$function)
     (forall (?r (cone ?r))
         (= (second ?r) ((SET.LIM$component ?r) 2)))
```

- The KIF function ‘`limiting-cone`’ that maps an opspan to its pullback (limiting pullback cone) (Figure 4). A limiting pullback cone is the special case of a general limiting cone over a pullback diagram (opspan).

```
(16) (KIF$function limiting-cone)
     (= (KIF$source limiting-cone) diagram)
     (= (KIF$target limiting-cone) cone)
     (KIF$restriction limiting-cone SET.LIM$limiting-cone)

(17) (KIF$function limit)
     (KIF$function pullback)
     (= pullback limit)
     (= (KIF$source limit) diagram)
     (= (KIF$target limit) SET$class)
     (forall (?d (diagram ?d))
         (= (limit ?d) (vertex (limiting-cone ?d))))

(18) (KIF$function projection1)
     (= (KIF$source projection1) diagram)
     (= (KIF$target projection1) SET.FTN$function)
     (forall (?d (GPH$diagram ?d)
         (= (projection1 ?d) (first (limiting-cone ?d))))

(19) (KIF$function projection2)
     (= (KIF$source projection2) diagram)
     (= (KIF$target projection2) SET.FTN$function)
     (forall (?d (GPH$diagram ?d)
         (= (projection2 ?d) (second (limiting-cone ?d))))
```

- We can show that the pullback class of an opspan is equal to the extent of the relation of the opspan, and the pullback projections are equal to the relational projections.

```
     (forall (?s (opspan ?s))
         (and (= (pullback ?s) (REL$extent (relation ?s)))
              (= (projection1 ?s) (REL$projection1 (relation ?s)))
              (= (projection2 ?s) (REL$projection2 (relation ?s)))))
```

- We can also show that the pullback class (first/second projection function) of one opspan is a subclass (restriction) of the pullback class (first/second projection function) of another opspan when the first opspan is a subopspan of the second opspan.

```
     (forall (?s1 (opspan ?s1) ?s2 (opspan ?s2))
         (=> (subopspan ?s1 ?s2)
             (and (SET$subclass (pullback ?s1) (pullback ?s2))
                  (SET.FTN$restriction (projection1 ?s1) (projection1 ?s2))
                  (SET.FTN$restriction (projection2 ?s1) (projection2 ?s2)))))
```

- There is a *mediator* function from the vertex of a pullback cone over a pullback diagram (opspan) to the pullback of the opspan. This is the unique function that commutes with the component functions of the cone. We have also introduced a <u>convenience term</u> ‘`pairing`’. Since the node class of the shape of opspan is the graph *three*, in order to define this via restriction of the general tupling function we must first define a tripling function. With an opspan parameter, the ternary KIF function ‘`(tripling ?d)`’ maps a triple of class functions that form a cone over the opspan to their mediator function. In applications, we can just use pairing, since the third function is redundant in any such triple, being the composition of either pairing component with the corresponding opspan component.

```
(20) (KIF$function mediator)
     (= (KIF$source mediator) cone)
     (= (KIF$target mediator) SET.FTN$function)
     (KIF$restriction mediator SET.LIM$mediator)

(21) (KIF$function tripling)
     (= (KIF$source tripling) diagram)
     (= (KIF$target tripling) KIF$partial-function)
     (forall (?d (diagram ?d))
         (and (= (KIF$source (tripling ?d))
                 (KIF$power [three SET.FTN$function]))
              (= (KIF$target (tripling ?d)) SET.FTN$function)
```

```
            (forall (?f1 ?f2 ?f3
                     ((KIF$power [three SET.FTN$function]) [?f1 ?f2 ?f3]))
                (<=> ((KIF$domain (tripling ?d)) [?f1 ?f2 ?f3])
                     (and (SET.FTN$function ?f1)
                          (SET.FTN$function ?f2)
                          (SET.FTN$function ?f3)
                          (= (SET.FTN$source ?f1) (SET.FTN$source ?f2))
                          (= (SET.FTN$source ?f2) (SET.FTN$source ?f3))
                          (= (SET.FTN$target ?f1) (class1 ?d))
                          (= (SET.FTN$target ?f2) (class2 ?d))
                          (= (SET.FTN$target ?f3) (opvertex ?d))
                          (= (SET.FTN$composition ?f1 (opfirst ?d)) ?f3)
                          (= (SET.FTN$composition ?f2 (opsecond ?d)) ?f3))))))
     (KIF$restriction tripling SET.LIM$tupling)
```

```
(22) (KIF$function pairing)
     (= (KIF$source pairing) diagram)
     (= (KIF$target pairing) KIF$partial-function)
     (forall (?d (diagram ?d))
         (and (= (KIF$source (pairing ?d)) (KIF$power [two SET.FTN$function]))
              (= (KIF$target (pairing ?d)) SET.FTN$function)
              (forall (?f1 (SET.FTN$function ?f1)
                       ?f2 (SET.FTN$function ?f2))
                  (and (<=> ((KIF$domain (pairing ?d)) [?f1 ?f2])
                            (and (= (SET.FTN$source ?f1) (SET.FTN$source ?f2))
                                 (= (SET.FTN$composition ?f1 (opfirst ?d))
                                    (SET.FTN$composition ?f2 (opsecond ?d)))))
                       (= ((pairing ?d) [?f1 ?f2])
                          ((tripling ?d)
                              [?f1 ?f2 (SET.FTN$composition ?f1 (opfirst ?d))]))))))
```

- A KIF ‘`binary-product-opspan`’ function maps a pair (of classes) to an associated pullback opspan. The opvertex is the *terminal* class, and the opfirst and opsecond functions are the *unique* functions for the pair of classes.

```
(23) (KIF$function binary-product-opspan)
     (= (KIF$source binary-product-opspan) SET.LIM.PRD$diagram)
     (= (KIF$target binary-product-opspan) diagram)
     (forall (?p (SET.LIM.PRD$diagram ?p))
         (and (= (class1 (binary-product-opspan ?p)) (SET.LIM.PRD$class1 ?p))
              (= (class2 (binary-product-opspan ?p)) (SET.LIM.PRD$class2 ?p))
              (= (opvertex (binary-product-opspan ?p)) SET.LIM$terminal)
              (= (opfirst (binary-product-opspan ?p))
                 (SET.LIM$unique (SET.LIM.PRD$class1 ?p)))
              (= (opsecond (binary-product-opspan ?p))
                 (SET.LIM$unique (SET.LIM.PRD$class2 ?p)))))
```

- Using this opspan, we can show that the notion of a product could be based upon pullbacks and the terminal class. 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 (= (SET.LIM.PRD$binary-product ?p)
                 (pullback (binary-product-opspan ?p)))
              (= (SET.LIM.PRD$projection1 ?p)
                 (projection1 (binary-product-opspan ?p)))
              (= (SET.LIM.PRD$projection2 ?p)
                 (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 (SET.LIM.PRD$diagram ?p))
         (= (SET.LIM.PRD$pairing ?p)
            (pairing (binary-product-opspan ?p))))
```

- 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 (for pullbacks).

```
(24) (KIF$function tau-cone)
```

```
        (= (KIF$source tau-cone) opspan)
        (= (KIF$target tau-cone) cone)
        (forall (?s (opspan ?s))
            (and (= (cone-diagram (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)))))

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

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

```
     (forall (?s (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)))))
```

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

```
(26) (KIF$function kernel-pair-diagram)
     (= (KIF$source kernel-pair-diagram) SET.FTN$function)
     (= (KIF$target kernel-pair-diagram) 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)))

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

Pullback Fibers

The following terms associated with pullback fibers are <u>convenience terms</u>.

$$\begin{tikzcd}
 & & \phi^S(b) \arrow[dll, "\pi^S_1"'] \arrow[d, "\iota^S_b"] \arrow[drr, "\pi^S_2"] & & \\
\phi^S_1(b) \arrow[dr, "\iota^S_{1b}"'] & & A_1 \times_B A_2 \arrow[dl, "1^{st}"'] \arrow[dr, "2^{nd}"] & & \phi^S_2(b) \arrow[dl, "\iota^S_{2b}"] \\
 & A_1 \arrow[dr, "f_1"'] & & A_2 \arrow[dl, "f_2"] & \\
 & & B & &
\end{tikzcd}$$

Figure 5: Pullback Fibers

- Associated with any pullback diagram (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 (Figure 5)
  - five fiber functions, the last two of which are derived,

$$\phi^S : B \to \wp(A_1 \times_B A_2)$$

$$\phi^S{}_1 : B \to \wp A_1 \qquad \phi^S{}_{12} : A_1 \to \wp A_2$$

$$\phi^S{}_2 : B \to \wp A_2 \qquad \phi^S{}_{21} : A_2 \to \wp A_1$$

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

$$\iota^S{}_b : \phi^S(b) \to A_1 \times_B A_2$$

$$\iota^S{}_{1b} : \phi^S{}_1(b) \to A_1 \qquad \iota^S{}_{12a1} : \phi^S{}_{12}(a_1) = \phi_2{}^S(f_1(a_1)) \to \phi^S(f_1(a_1))$$

$$\iota^S{}_{2b} : \phi^S{}_2(b) \to A_2 \qquad \iota^S{}_{21a2} : \phi^S{}_{21}(a_2) = \phi_1{}^S(f_2(a_2)) \to \phi^S(f_2(a_2))$$

  - and two projection functionals

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

Here are the pointwise definitions.

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

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

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

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

```
(28) (KIF$function fiber)
     (= (KIF$source fiber) opspan)
     (= (KIF$target fiber) SET.FTN$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))))))

(29) (KIF$function fiber1)
     (= (KIF$source fiber1) opspan)
     (= (KIF$target fiber1) SET.FTN$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)))))

(30) (KIF$function fiber2)
     (= (KIF$source fiber2) opspan)
     (= (KIF$target fiber2) SET.FTN$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)))))

(31) (KIF$function fiber12)
     (= (KIF$source fiber12) opspan)
     (= (KIF$target fiber12) SET.FTN$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 ?s)))))

(32) (KIF$function fiber21)
     (= (KIF$source fiber21) opspan)
     (= (KIF$target fiber21) SET.FTN$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 ?s)))))

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

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

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

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

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

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

(39) (KIF$function fiber2-projection)
```

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

Subequalizers

SET.LIM.SEQU

A *subequalizer* (Figure 6) is a lax equalizer – a lax limit for a lax diagram consisting of a parallel pair of functions whose target is an order.

$$\begin{tikzcd} S \arrow[r, "\sigma"] \arrow[dr] & A \arrow[d, shift right, "f_1"'] \arrow[d, shift left, "f_2"] \\ & \boldsymbol{B} \end{tikzcd}$$

Figure 6: Subequalizer

- 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) (KIF$collection lax-diagram)
    (KIF$collection lax-parallel-pair)
    (= lax-parallel-pair lax-diagram)

(2) (KIF$function order)
    (= (KIF$source order) lax-diagram)
    (= (KIF$target order) ORD$preorder)

(3) (KIF$function source)
    (= (KIF$source source) lax-diagram)
    (= (KIF$target source) SET$class)

(4) (KIF$function function1)
    (= (KIF$source function1) lax-diagram)
    (= (KIF$target function1) SET.FTN$function)

(5) (KIF$function function2)
    (= (KIF$source function2) lax-diagram)
    (= (KIF$target function2) SET.FTN$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) (KIF$function lax)
    (= (KIF$source lax) SET.LIM.EQU$diagram)
    (= (KIF$target lax) lax-diagram)
    (forall (?p (SET.LIM.EQU$diagram ?p))
        (and (= (order (lax ?p)) (ORD$identity (SET.LIM.EQU$target ?p)))
             (= (source (lax ?p)) (SET.LIM.EQU$source ?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) (KIF$function parallel-pair)
    (= (KIF$source parallel-pair) lax-diagram)
    (= (KIF$target parallel-pair) SET.LIM.EQU$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))))
```

- The underlying parallel pair of the laxation of a parallel pair is itself.

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

- Lax parallel pairs are determined by their order and crisp parallel pair.

```
(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 subequalizers. Each subequalizer cone has a base *lax diagram*, a *vertex* class, and a function called *function* whose source class is the vertex and whose target class is the source class of the parallel-pair. A subequalizer cone is the very special case of a lax cone over a 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) (KIF$collection lax-cone)

(9) (KIF$function lax-cone-diagram)
    (= (KIF$source lax-cone-diagram) lax-cone)
    (= (KIF$target lax-cone-diagram) lax-diagram)

(10) (KIF$function vertex)
    (= (KIF$source vertex)lax-cone)
    (= (KIF$target vertex)SET$class)

(11) (KIF$function function)
     (= (KIF$source function) lax-cone)
     (= (KIF$target function) SET.FTN$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) ?x ((vertex ?r) ?x))
         ((order (lax-cone-diagram ?r))
             ((function1 (lax-cone-diagram ?r)) ((function ?r) ?x))
             ((function2 (lax-cone-diagram ?r)) ((function ?r) ?x))))
```

- The KIF function ‘`limiting-lax-cone`’ maps a diagram (lax-parallel-pair $f_1, f_2 : A \rightarrow \boldsymbol{B} = \langle B, \leq \rangle$) to its subequalizer (lax limiting subequalizer cone) (Figure 6). This asserts that a subequalizer exists for any diagram (lax-parallel-pair). The universality of this lax-limit is expressed by axioms for the mediator function. The vertex of the subequalizer cone is a specific lax limit class $\{a \in A \mid f_1(a) \leq f_2(a)\} \subseteq A$ given by the KIF function ‘`subequalizer`’. It comes equipped with a canonical subequalizing function ‘`subcanon`’, which is the inclusion of the subequalizer class into source class A. This notation is for convenience of reference. 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) (KIF$function limiting-lax-cone)
     (= (KIF$source limiting-lax-cone) lax-diagram)
     (= (KIF$target limiting-lax-cone) lax-cone)
     (forall (?d (lax-diagram ?d))
        (= (lax-cone-diagram (limiting-lax-cone ?d)) ?d))

(13) (KIF$function lax-limit)
     (KIF$function subequalizer)
     (= subequalizer lax-limit)
     (= (KIF$source subequalizer) lax-diagram)
     (= (KIF$target subequalizer) SET$class)
     (forall (?d (lax-diagram ?d))
        (= (subequalizer ?d)
           (vertex (limiting-lax-cone ?d))))
```

```
(14) (KIF$function subcanon)
     (= (KIF$source subcanon) lax-diagram)
     (= (KIF$target subcanon) SET.FTN$function)
     (forall (?d (lax-diagram ?d))
        (= (subcanon ?d) (function (limiting-lax-cone ?d))))

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

o 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 subcanon and lax-cone function. We use a KIF definite description to define this. Existence and uniqueness represents the universality of the subequalizer operator.

```
(15) (KIF$function mediator)
     (= (KIF$source mediator) lax-cone)
     (= (KIF$target mediator) SET.FTN$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))))))
```

Colimits

SET.COL

Here we present axioms that make the quasicategory of classes and functions cocomplete. The finite colimits in the IFF Classification Ontology use this. Colimits are dual to limits. We assert the existence of initial classes, binary coproducts, coequalizers of parallel pairs of functions, and pushouts of spans. All are defined to be specific classes – for example, the binary coproduct is the disjoint union. Because of commonality, the terminology for binary coproducts, coequalizers and pushouts are put into sub-namespaces. This commonality has been abstracted into a general formulation of colimits. The *diagrams* and *colimits* are denoted by both generic and specific terminology. A *colimit* is the opvertex of a colimiting diagram of a certain shape. The base diagram for a colimit is represented by the diagram terminology in the (large) graph namespace in the IFF Category Theory Ontology.

$$
\begin{tikzcd}
\mathit{cocone}(G_k) \arrow[r] \arrow[d, shift left, "\mathit{base}_k"] & \mathit{cocone} \arrow[d, shift left, "\mathit{base}"] \\
\mathit{diagram}(G_k) \arrow[r] \arrow[u, shift left, "\mathit{colimit}_k"] & \mathit{diagram} \arrow[u, shift left, "\mathit{colimit}"] \arrow[d, "\mathit{shape}"] \\
 & \mathit{graph}
\end{tikzcd}
$$

Diagram 5: Diagrams, Cocones and Fibers

- A *cocone* (Diagram 6) consists of a base *diagram*, an *opvertex*, and a collection of *component* functions indexed by the nodes in the shape of the diagram. The cocone is situated under the base diagram. The component functions form commutative diagrams with the diagram functions.

```
(1) (KIF$collection cocone)

(2) (KIF$function cocone-diagram)
    (KIF$function base)
    (= cocone-diagram base)
    (= (KIF$source cocone-diagram) cocone)
    (= (KIF$target cocone-diagram) GPH$diagram)

(3) (KIF$function opvertex)
    (= (KIF$source opvertex) cocone)
    (= (KIF$target opvertex) SET$class)
```

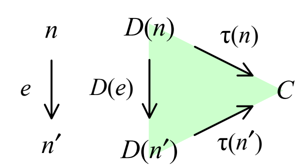

Diagram 6: Cocone

```
(4) (KIF$function component)
    (= (KIF$source component) cocone)
    (= (KIF$target component) KIF$function)
    (forall (?s (cocone ?s))
        (and (= (KIF$source (component ?s)) (GPH$node (GPH$shape (cocone-diagram ?s))))
             (= (KIF$target (component ?s)) SET.FTN$function)
             (forall (?n ((GPH$node (GPH$shape (cocone-diagram ?s))) ?n))
                 (and (= (SET.FTN$source ((component ?s) ?n))
                         ((GPH$class (cocone-diagram ?s)) ?n))
                      (= (SET.FTN$target ((component ?s) ?n)) (opvertex ?s))))
             (forall (?e ((GPH$edge (GPH$shape (cocone-diagram ?s))) ?e))
                 (= (SET.FTN$composition
                        ((GPH$function (cocone-diagram ?s)) ?e)
                        ((component ?s) ((GPH$target (GPH$shape (cocone-diagram ?s))) ?e)))
                    ((component ?s) ((GPH$source (GPH$shape (cocone-diagram ?s))) ?e))))))
```

- The *cocone-fiber* $\mathit{cocone}(G)$ of any graph G is the collection of all cocones whose base diagram has shape G.

```
(5) (KIF$function base-shape)
    (= (KIF$source base-shape) cocone)
    (= (KIF$target base-shape) GPH$graph)
    (forall (?s (cocone ?s))
        (= (base-shape ?s) (GPH$shape (base ?s))))

(6) (KIF$function cocone-fiber)
    (= (KIF$source cocone-fiber) GPH$graph))
    (= (KIF$target cocone-fiber) KIF$collection)
    (= cocone-fiber (KIF$fiber base-shape))
```

- The KIF function '`colimiting-cocone`' maps a diagram to its colimit (colimiting cocone) (Diagram 7). This asserts that a colimit exists for any diagram. The universality of this colimit is expressed by axioms for the comediator function. The opvertex of the colimiting cocone is a specific *colimit* class given by the KIF function ‘`colimit`’. It comes equipped with component injection functions. This notation is for convenience of reference. Axiom (#) ensures that this colimit is specific, the disjoint union. Axiom (%) ensures that the component injection functions are also specific, the injections into the disjoint union.

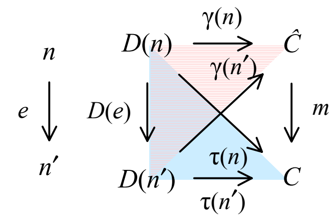

Diagram 7: Colimiting Cocone

```
(7) (KIF$function colimiting-cocone)
    (= (KIF$source colimiting-cocone) GPH$diagram)
    (= (KIF$target colimiting-cocone) cocone)
    (forall (?d (GPH$diagram ?d))
        (= (cocone-diagram (colimiting-cocone ?d)) ?d))

(8) (KIF$function colimit)
    (= (KIF$source colimit) GPH$diagram)
    (= (KIF$target colimit) SET$class)
    (forall (?d (GPH$diagram ?d))
        (= (colimit ?d) (opvertex (colimiting-cocone ?d))))
(#) (forall (?d (GPH$diagram ?d))
        (SET$subclass (colimit ?d) (KIF$coproduct (GPH$class ?d))))

(9) (KIF$function injection)
    (= (KIF$source injection) GPH$diagram)
    (= (KIF$target injection) KIF$function)
    (forall (?d (GPH$diagram ?d))
        (and (= (KIF$source (injection ?d)) (GPH$node (GPH$shape ?d)))
             (= (KIF$target (injection ?d)) SET.FTN$function)
             (forall (?n ((GPH$node (GPH$shape ?d)) ?n))
                 (and (= (SET.FTN$source ((injection ?d) ?n)) ((GPH$class ?d) ?n))
                      (= (SET.FTN$target ((injection ?d) ?n)) (colimit ?d))
                      (= ((injection ?d) ?n)
                         ((component (colimiting-cocone ?d)) ?n))))))
(%) (forall (?d (GPH$diagram ?d)
             ?n ((GPH$node (GPH$shape ?d)) ?n)
             ?x (((GPH$class ?d) ?n) ?x))
        (= (((injection ?d) ?n) ?x) [?n ?x]))
```

- There is a *comediator* function from the colimit of a diagram to the opvertex of a cocone under the diagram. This is the unique function that commutes with the component functions of the cocone. We use a KIF definite description to define this. Existence and uniqueness represents the universality of the colimit operator. We have also introduced a <u>convenience term</u> ‘`cotupling`’. With a diagram parameter, the KIF function ‘`(cotupling ?d)`’ maps a tuple of class functions, that form a cocone under the diagram, to their comediator (cotupling) function.

```
(10) (KIF$function comediator)
     (= (KIF$source comediator) cocone)
     (= (KIF$target comediator) SET.FTN$function)
     (forall (?s (cocone ?s))
        (= (comediator ?s)
           (the (?f (SET.FTN$function ?f))
                (and (= (SET.FTN$source ?f) (colimit (cocone-diagram ?s)))
                     (= (SET.FTN$target ?f) (opvertex ?s))
                     (forall (?n ((GPH$node (GPH$shape (cocone-diagram ?s))) ?n))
                         (= (SET.FTN$composition
                                ((injection (cocone-diagram ?s)) ?n) ?f)
                            ((component ?s) ?n)))))))

(11) (KIF$function cotupling-cocone)
     (KIF$source cotupling-cocone) GPH$diagram)
     (KIF$target cotupling-cocone) KIF$partial-function)
     (forall (?d (GPH$diagram ?d))
         (and (= (KIF$source (cotupling-cocone ?d))
                 (KIF$power [(GPH$node (GPH$shape ?d)) SET.FTN$function]))
              (= (KIF$target (cotupling-cocone ?d)) cocone)
              (forall (?f ((KIF$power [(GPH$node (GPH$shape ?d)) SET.FTN$function]) ?f))
                  (<=> ((KIF$domain (cotupling-cocone ?d)) ?f)
                       (and (forall (?n ((GPH$node (GPH$shape ?d)) ?n))
                                (= (SET.FTN$source (?f ?n)) ((GPH$class ?d) ?n)))
                            (exists (?c (collection ?c))
```

```
                                   (forall (?n ((GPH$node (GPH$shape ?d)) ?n))
                                       (= (SET.FTN$target (?f ?n)) ?c)))
                               (forall (?e ((GPH$edge (GPH$shape ?d)) ?e))
                                   (= (SET.FTN$composition
                                          ((GPH$function ?d) ?e)
                                          (?f ((GPH$source (GPH$shape ?d)) ?e)))
                                      (?f ((GPH$target (GPH$shape ?d)) ?e))))))))
         (forall (?d (GPH$diagram ?d)
                  ?f ((KIF$domain (cotupling-cocone ?d)) ?f))
             (and (= (cocone-diagram ((cotupling-cocone ?d) ?f)) ?d)
                  (forall (?n ((GPH$node (GPH$shape ?d)) ?n))
                      (and (= (opvertex ((cotupling-cocone ?d) ?f)) (SET.FTN$target (?f ?n)))
                           (= ((component ((cotupling-cocone ?d) ?f)) ?n) (?f ?n))))))

    (12) (KIF$function cotupling)
         (= (KIF$source cotupling) GPH$diagram)
         (= (KIF$target cotupling) KIF$partial-function)
         (forall (?d (GPH$diagram ?d))
             (and (= (KIF$source (cotupling ?d))
                     (KIF$power [(GPH$node (GPH$shape ?d)) SET.FTN$function]))
                  (= (KIF$target (cotupling ?d)) SET.FTN$function)
                  (= (KIF$domain (cotupling ?d)) (KIF$domain (cotupling-cocone ?d)))
                  (forall ?f ((KIF$domain (cotupling ?d)) ?f))
                      (= ((cotupling ?d) ?f)
                         (comediator ((cotupling-cocone ?d) ?f))))))
```

The Initial Class

- A cocone, whose base diagram is the diagram of empty shape, is essentially just a class – the opvertex class of the cocone. There is an isomorphism between cocones under the empty diagram and classes.

```
    (SET.FTN$isomorphic (SET.LIM$cone-fiber GPH$empty) SET$class)
```

- The colimit (there is only one, since there is only one diagram) is special.

```
(13) (SET$class initial)
     (SET$class null)
     (= initial null)
     (=initial (colimit GPH$empty-diagram))
```

- The comediator of any class (cocone) is the unique function to that class from the initial class. Therefore, the colimit is the null or initial class. For each class C there is a *counique* function $!_C : 1 \rightarrow C$ from the null class.

```
(14) (KIF$function counique)
     (= (KIF$source counique) SET$class)
     (= (KIF$target counique) SET.FTN$function)
     (forall (?c (SET$class ?c))
         (= (counique ?c)
            (the (?f (SET.FTN$function ?f))
                 (and (= (SET.FTN$source ?f) null)
                      (= (SET.FTN$target ?f) ?c)))))
```

- The following facts can be proven: the colimit is the initial class, and the comediator is the counique function.

```
     (= initial SET$initial)
     (= counique SET.FTN$counique)
```

Binary Coproducts

SET.COL.COPRD

A *binary coproduct* (Figure 7) is a finite limit for a diagram of shape *two* = • •. Such a diagram (of classes and functions) is called a *pair* of classes.

$$\begin{tikzcd} C_1 \arrow[dr, "\iota_1"'] & & C_2 \arrow[dl, "\iota_2"] \\ & C_1{+}C_2 & \end{tikzcd}$$

Figure 7: Binary Coproduct

- A *pair* (of classes) is the appropriate base diagram for a binary coproduct. Each pair consists of a pair of classes called *class1* and *class2*. We use either the generic term '`diagram`' or the specific term '`pair`' to denote the *pair* collection. A pair is the special case of a general diagram of shape *two*.

```
(1) (KIF$collection diagram)
    (KIF$collection pair)
    (= pair diagram)
    (KIF$subcollection diagram GPH$diagram)
    (= diagram (GPH$diagram-fiber GPH$two))

(2) (KIF$function class1)
    (= (KIF$source class1) diagram)
    (= (KIF$target class1) SET$class)
    (forall (?d (diagram ?d))
        (= (class1 ?d) ((GPH$class ?d) 1)))

(3) (KIF$function class2)
    (= (KIF$source class2) diagram)
    (= (KIF$target class2) SET$class)
    (forall (?d (diagram ?d))
        (= (class2 ?d) ((GPH$class ?d) 2)))
```

- By the unique determination inherited from the general case, we can prove the isomorphism *pair* ≅ *class*×*class*.

```
    (KIF$isomorphic pair (KIF$binary-product [SET$class SET$class]))
```

- Every pair has an opposite.

```
(4) (KIF$function opposite)
    (= (KIF$source opposite) pair)
    (= (KIF$target opposite) 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 *binary coproduct cocone* consists of a pair of functions called *opfirst* and *opsecond*. These are required to have a common target class called the *opvertex* of the cocone. Each binary coproduct cocone is situated under a binary coproduct diagram (pair). A binary coproduct cocone is the special case of a general cocone under a binary coproduct diagram (pair of classes).

```
(5) (KIF$collection cocone)
    (KIF$subcollection cocone SET.COL$cocone)
    (= cocone (SET.COL$cocone-fiber GPH$two))

(6) (KIF$function cocone-diagram)
    (= (KIF$source cocone-diagram) cocone)
    (= (KIF$target cocone-diagram) diagram)
    (SET.FTN$restriction cocone-diagram SET.COL$cocone-diagram)

(7) (KIF$function opvertex)
    (= (KIF$source opvertex) cocone)
    (= (KIF$target opvertex) SET$class)
    (SET.FTN$restriction opvertex SET.COL$opvertex)

(8) (KIF$function opfirst)
    (= (KIF$source opfirst) cocone)
    (= (KIF$target first) SET.FTN$function)
    (forall (?s (cocone ?s))
        (= (opfirst ?s) ((SET.COL$component ?s) 1)))

(9) (KIF$function opsecond)
    (= (KIF$source opsecond) cocone)
    (= (KIF$target opsecond) SET.FTN$function)
    (forall (?s (cocone ?s))
        (= (opsecond ?s) ((SET.COL$component ?s) 2)))
```

- The KIF function '`colimiting-cocone`' maps a pair of classes to its binary coproduct (colimiting binary coproduct cocone) (Figure 7). A colimiting binary coproduct cocone is the special case of a general colimiting cocone over a binary coproduct diagram (pair of classes).

```
(10) (KIF$function colimiting-cocone)
     (= (KIF$source colimiting-cocone) diagram)
     (= (KIF$target colimiting-cocone) cocone)
     (KIF$restriction colimiting-cocone SET.COL$colimiting-cocone)

(11) (KIF$function colimit)
     (KIF$function binary-coproduct)
     (= binary-coproduct colimit)
     (= (KIF$source colimit) diagram)
     (= (KIF$target colimit) SET$class)
     (forall (?d (diagram ?d))
         (= (colimit ?d) (opvertex (colimiting-cocone ?d))))

(12) (KIF$function injection1)
     (= (KIF$source injection1) diagram)
     (= (KIF$target injection1) SET.FTN$function)
     (forall (?d (GPH$diagram ?d)
         (= (injection1 ?d) (opfirst (colimiting-cocone ?d))))

(13) (KIF$function injection2)
     (= (KIF$source injection2) diagram)
     (= (KIF$target injection2) SET.FTN$function)
     (forall (?d (GPH$diagram ?d)
         (= (injection2 ?d) (opsecond (colimiting-cocone ?d))))
```

- There is a *comediator* function to the opvertex of a binary coproduct cocone over a binary coproduct diagram (pair of classes) from the binary coproduct of the pair. This is the unique function that commutes with the component functions of the cocone. We have also introduced a "convenience term" copairing. With a diagram parameter, this maps a pair of class functions, which form a binary coproduct cocone with the diagram, to their comediator (or *copairing*) function.

```
(14) (KIF$function comediator)
     (= (KIF$source comediator) cocone)
     (= (KIF$target comediator) SET.FTN$function)
     (KIF$restriction comediator SET.COL$comediator)

(15) (KIF$function copairing)
     (= (KIF$source copairing) diagram)
     (= (KIF$target copairing) KIF$partial-function)
     (forall (?d (diagram ?d))
         (and (= (KIF$source (copairing ?d))
                 (KIF$power [two SET.FTN$function]))
              (= (KIF$target (copairing ?d)) SET.FTN$function)
              (forall (?f1 ?f2 ((KIF$power [two SET.FTN$function]) [?f1 ?f2]))
                  (<=> ((KIF$domain (copairing ?d)) [?f1 ?f2])
                       (and (SET.FTN$function ?f1)
                            (SET.FTN$function ?f2)
                            (= (SET.FTN$target?f1) (SET.FTN$target ?f2))
                            (= (SET.FTN$source ?f1) (class1 ?d))
                            (= (SET.FTN$source ?f2) (class2 ?d)))))))
     (KIF$restriction copairing SET.COL$cotupling)
```

- The coproduct of the opposite of a pair is isomorphic to the coproduct of the pair. This isomorphism is mediated by the *tau* or *twist* function (for coproducts).

```
(16) (KIF$function tau-cocone)
     (= (KIF$source tau-cocone) pair)
     (= (KIF$target tau-cocone) cocone)
     (forall (?p (pair ?p))
        (and (= (cocone-diagram (tau-cocone ?p)) ?p)
             (= (opvertex (tau-cocone ?p)) (binary-coproduct (opposite ?p)))
             (= (opfirst (tau-cocone ?p)) (injection2 (opposite ?p)))
             (= (opsecond (tau-cocone ?p)) (injection1 (opposite ?p)))))

(17) (KIF$function tau)
```

```
        (= (KIF$source tau) pair)
        (= (KIF$target tau) SET.FTN$function)
        (forall (?p (pair ?p))
           (and (= (SET.FTN$source (tau ?p)) (binary-coproduct ?p))
                (= (SET.FTN$target (tau ?p)) (binary-coproduct (opposite ?p)))
                (= (tau ?p) (comediator (tau-cocone ?p)))))
```

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

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

Coequalizers

`SET.COL.COEQ`

A *coequalizer* (Figure 8) is a finite `colimit` for a diagram of shape *parallel-pair* = $\cdot \rightrightarrows \cdot$. Such a diagram (of classes and functions) is called a *parallel pair* of functions.

$$\begin{tikzcd} C_1 \arrow[d, shift right, "f_1"'] \arrow[d, shift left, "f_2"] \arrow[dr] & \\ C_2 \arrow[r, "\rho"'] & E \end{tikzcd}$$

Figure 8: Coequalizer

- A *parallel pair* is the appropriate base diagram for a coequalizer. Each parallel pair consists of a pair of functions called *funtion1* and *function2* that share the same *source* and *target* classes. We use either the generic term '`diagram`' or the specific term '`parallel-pair`' to denote the *parallel pair* collection. A parallel pair is a special case of a general diagram of shape *parallel-pair*.

```
(1) (KIF$collection diagram)
    (KIF$collection parallel-pair)
    (= parallel-pair diagram)
    (KIF$subcollection diagram GPH$diagram)
    (= diagram (GPH$diagram-fiber GPH$parallel-pair))

(2) (KIF$function source)
    (= (KIF$source source) diagram)
    (= (KIF$target source) SET$class)
    (forall (?d (diagram ?d))
        (= (source ?d) ((GPH$class ?d) 1)))

(3) (KIF$function target)
    (= (KIF$source target) diagram)
    (= (KIF$target target) SET$class)
    (forall (?d (diagram ?d))
        (= (target ?d) ((GPH$class ?d) 2)))

(4) (KIF$function function1)
    (= (KIF$source function1) diagram)
    (= (KIF$target function1) SET.FTN$function)
    (forall (?d (diagram ?d))
        (= (function1 ?d) ((GPH$function ?d) 1)))

(5) (KIF$function function2)
    (= (KIF$source function2) diagram)
    (= (KIF$target function2) SET.FTN$function)
    (forall (?d (diagram ?d))
        (= (function2 ?d) ((GPH$function ?d) 2)))
```

- A *coequalizer cocone* consists of an *opvertex* class and a function called *function* whose target class is the opvertex and whose source class is the target class of the functions in the parallel-pair. Each coequalizer cocone is situated under a coequalizer diagram (parallel pair of functions). A coequalizer cocone is the special case of a general *cocone* under a coequalizer diagram.

```
(6) (KIF$collection cocone)
    (KIF$subcollection cocone SET.COL$cocone)
    (= cocone (SET.COL$ cocone-fiber GPH$parallel-pair))

(7) (KIF$function cocone-diagram)
    (= (KIF$source cocone-diagram) cocone)
    (= (KIF$target cocone-diagram) diagram)
```

```
    (SET.FTN$restriction cocone-diagram SET.COL$cocone-diagram)
```

```
(8) (KIF$function opvertex)
    (= (KIF$source opvertex) cocone)
    (= (KIF$target opvertex) SET$class)
    (SET.FTN$restriction opvertex SET.COL$opvertex)
```

```
(9) (KIF$function function)
    (= (KIF$source function) cocone)
    (= (KIF$target function) SET.FTN$function)
    (forall (?s (cocone ?s))
        (= (function ?s) ((SET.COL$component ?s) 2)))
```

- The KIF function ‘`colimiting-cocone`’ maps a parallel pair of functions to its coequalizer (colimiting coequalizer cocone) (Figure 8). A colimiting coequalizer cocone is the special case of a general colimiting cocone over a coequalizer diagram (parallel pair of functions).

```
(10) (KIF$function colimiting-cocone)
     (= (KIF$source colimiting-cocone) diagram)
     (= (KIF$target colimiting-cocone) cocone)
     (KIF$restriction colimiting-cocone SET.COL$colimiting-cocone)
```

```
(11) (KIF$function colimit)
     (KIF$function coequalizer)
     (= coequalizer colimit)
     (= (KIF$source colimit) diagram)
     (= (KIF$target colimit) SET$class)
     (forall (?d (diagram ?d))
         (= (colimit ?d)
            (opvertex (colimiting-cocone ?d))))
```

```
(12) (KIF$function canon)
     (= (KIF$source canon) diagram)
     (= (KIF$target canon) SET.FTN$function)
     (forall (?d (diagram ?d))
         (= (canon ?d) (function (colimiting-cocone ?p))))
```

- There is a *comediator* function to the opvertex of an coequalizer cocone under a coequalizer diagram (parallel pair of functions) from the coequalizer of the parallel pair. This is the unique function that commutes with the component functions of the cocone.

```
(13) (KIF$function comediator)
     (= (KIF$source comediator) cocone)
     (= (KIF$target comediator) SET.FTN$function)
     (KIF$restriction comediator SET.COL$comediator)
```

Pushouts

SET.COL.PSH

A *pushout* (Figure 9) is a finite `colimit` for a diagram of shape *span* = $\bullet \leftarrow \bullet \rightarrow \bullet$. Such a diagram (of classes and functions) is called an *span*.

$$\begin{tikzcd}
& C_3 \arrow[dl, "f_1"'] \arrow[dr, "f_2"] & \\
C_1 \arrow[dr, "\iota_1"'] & & C_2 \arrow[dl, "\iota_2"] \\
& C_1 \times_{C_3} C_2 &
\end{tikzcd}$$

Figure 9: Pushout

- A *span* is the appropriate base diagram for a pushout. Each span consists of a pair of functions called *first* and *second*. These are required to have a common source class, denoted as the *vertex*. We use either the generic term ‘`diagram`’ or the specific term ‘`span`’ to denote the *span* collection. A span is the special case of a general diagram whose shape is the graph that is also named *span*.

```
(1) (KIF$collection diagram)
    (KIF$collection span)
    (= span diagram)
    (KIF$subcollection diagram GPH$diagram)
    (= diagram (GPH$diagram-fiber GPH$span))
```

```
(2) (KIF$function class1)
    (= (KIF$source class1) diagram)
    (= (KIF$target class1) SET$class)
    (forall (?d (diagram ?d))
        (= (class1 ?d) ((GPH$class ?d) 1)))
```

```
(3) (KIF$function class2)
    (= (KIF$source class2) diagram)
    (= (KIF$target class2) SET$class)
    (forall (?d (diagram ?d))
        (= (class2 ?d) ((GPH$class ?d) 2)))

(4) (KIF$function vertex)
    (= (KIF$source vertex) diagram)
    (= (KIF$target vertex) SET$class)
    (forall (?d (diagram ?d))
        (= (vertex ?d) ((GPH$class ?d) 3)))

(5) (KIF$function first)
    (= (KIF$source first) diagram)
    (= (KIF$target first) SET.FTN$function)
    (forall (?d (diagram ?d))
        (= (first ?d) ((GPH$function ?d) 1)))

(6) (KIF$function second)
    (= (KIF$source second) diagram)
    (= (KIF$target second) SET.FTN$function)
    (forall (?d (diagram ?d))
        (= (second ?d) ((GPH$function ?d) 2)))
```

- The *pair* of source classes (suffixing discrete diagram) of any span (pushout diagram) is named.

```
(7) (KIF$function pair)
    (= (KIF$source pair) diagram)
    (= (KIF$target pair) SET.LIM.PRD$diagram)
    (= pair
       (GPH$restriction [GPH$two GPH$span]))
```

- Every span has an opposite.

```
(8) (KIF$function opposite)
    (= (KIF$source opposite) span)
    (= (KIF$target opposite) span)
    (forall (?r (span ?r))
        (and (= (class1 (opposite ?r)) (class2 ?r))
             (= (class2 (opposite ?r)) (class1 ?r))
             (= (vertex (opposite ?r)) (vertex ?r))
             (= (first (opposite ?r)) (second ?r))
             (= (second (opposite ?r)) (first ?r))))
```

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

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

- A *pushout cocone* consists of an overlying pushout diagram (*span*), an *opvertex* class, and a pair of functions called *opfirst* and *opsecond*, whose common target class is the opvertex and whose source classes are the target classes of the functions in the span. The opfirst and opsecond functions form a commutative diagram with the span. Each pushout cocone is situated under its overlying pushout diagram (span). A pushout cocone is the special case of a general cocone under a pushout diagram (span).

```
(9) (KIF$collection cocone)
    (KIF$subcollection cocone SET.COL$cocone)
    (= cocone (SET.COL$cocone-fiber GPH$span))

(10) (KIF$function cocone-diagram)
     (= (KIF$source cocone-diagram) cocone)
     (= (KIF$target cocone-diagram) diagram)
     (SET.FTN$restriction cocone-diagram SET.COL$cocone-diagram)

(11) (KIF$function opvertex)
     (= (KIF$source opvertex) cocone)
     (= (KIF$target opvertex) SET$class)
     (SET.FTN$restriction opvertex SET.COL$opvertex)

(12) (KIF$function opfirst)
```

```
     (= (KIF$source opfirst) cocone)
     (= (KIF$target opfirst) SET.FTN$function)
     (forall (?s (cocone ?s))
         (= (opfirst ?s) ((SET.COL$component ?s) 1)))

(13) (KIF$function opsecond)
     (= (KIF$source opsecond) cocone)
     (= (KIF$target opsecond) SET.FTN$function)
     (forall (?s (cocone ?s))
         (= (opsecond ?s) ((SET.COL$component ?s) 2)))
```

- The KIF function '`colimiting-cocone`' that maps an span to its pushout (colimiting pushout cocone) (Figure 9). A colimiting pushout cocone is the special case of a general colimiting cocone over a pushout diagram (span).

```
(14) (KIF$function colimiting-cocone)
     (= (KIF$source colimiting-cocone) diagram)
     (= (KIF$target colimiting-cocone) cocone)
     (KIF$restriction colimiting-cocone SET.COL$colimiting-cocone)

(15) (KIF$function colimit)
     (KIF$function pushout)
     (= pushout colimit)
     (= (KIF$source colimit) diagram)
     (= (KIF$target colimit) SET$class)
     (forall (?d (diagram ?d))
         (= (colimit ?d) (opvertex (colimiting-cocone ?d))))

(16) (KIF$function injection1)
     (= (KIF$source injection1) diagram)
     (= (KIF$target injection1) SET.FTN$function)
     (forall (?d (GPH$diagram ?d)
         (= (injection1 ?d) (opfirst (colimiting-cocone ?d))))

(17) (KIF$function injection2)
     (= (KIF$source injection2) diagram)
     (= (KIF$target injection2) SET.FTN$function)
     (forall (?d (GPH$diagram ?d)
         (= (injection2 ?d) (opsecond (colimiting-cocone ?d))))
```

- There is a *comediator* function to the opvertex of a pushout cocone under a pushout diagram (span) from the pushout of the span. This is the unique function that commutes with the component functions of the cocone. We have also introduced a <u>convenience term</u> '`copairing`'. Since the node class of the shape of span is the graph *three*, in order to define this via restriction of the general tupling function we must first define a cotripling function. With a span parameter, the ternary KIF function '`(cotripling ?d)`' maps a triple of class functions that form a cocone under the span to their comediator function. In applications, we can just use pairing, since the third function is redundant in any such triple, being the composition of either pairing component with the corresponding span component.

```
(18) (KIF$function comediator)
     (= (KIF$source comediator) cocone)
     (= (KIF$target comediator) SET.FTN$function)
     (KIF$restriction comediator SET.COL$comediator)

(19) (KIF$function cotripling)
     (= (KIF$source cotripling) diagram)
     (= (KIF$target cotripling) KIF$partial-function)
     (forall (?d (diagram ?d))
         (and (= (KIF$source (cotripling ?d))
                 (KIF$power [three SET.FTN$function]))
              (= (KIF$target (cotripling ?d)) SET.FTN$function)
              (forall (?f1 ?f2 ?f3
                       ((KIF$power [three SET.FTN$function]) [?f1 ?f2 ?f3]))
                  (<=> ((KIF$domain (cotripling ?d)) [?f1 ?f2 ?f3])
                       (and (SET.FTN$function ?f1)
                            (SET.FTN$function ?f2)
                            (SET.FTN$function ?f3)
                            (= (SET.FTN$target ?f1) (SET.FTN$target ?f2))
                            (= (SET.FTN$target ?f2) (SET.FTN$target ?f3))
```

```
                         (= (SET.FTN$source ?f1) (class1 ?d))
                         (= (SET.FTN$source ?f2) (class2 ?d))
                         (= (SET.FTN$source ?f3) (vertex ?d))
                         (= (SET.FTN$composition (first ?d) ?f1) ?f3)
                         (= (SET.FTN$composition (second ?d) ?f2) ?f3))))))
     (KIF$restriction cotripling SET.COL$cotupling)
```

```
(20) (KIF$function copairing)
     (= (KIF$source copairing) diagram)
     (= (KIF$target copairing) KIF$partial-function)
     (forall (?d (diagram ?d))
         (and (= (KIF$source (copairing ?d)) (KIF$power [two SET.FTN$function]))
              (= (KIF$target (copairing ?d)) SET.FTN$function)
              (forall (?f1 (SET.FTN$function ?f1)
                       ?f2 (SET.FTN$function ?f2))
                  (and (<=> ((KIF$domain (copairing ?d)) [?f1 ?f2])
                            (and (= (SET.FTN$target ?f1) (SET.FTN$target ?f2))
                                 (= (SET.FTN$composition (first ?d) ?f1)
                                    (SET.FTN$composition (second ?d) ?f2))))
                       (= ((copairing ?d) [?f1 ?f2])
                          ((cotripling ?d)
                              [?f1 ?f2 (SET.FTN$composition (first ?d) ?f1)]))))))
```

- A KIF ‘`binary-coproduct-span`’ function maps a pair (of classes) to an associated pushout opspan. The vertex is the *initial* class, and the first and second functions are the *counique* functions for the pair of classes.

```
(21) (KIF$function binary-coproduct-span)
     (= (KIF$source binary-coproduct-span) SET.COL.COPRD$diagram)
     (= (KIF$target binary-coproduct-span) diagram)
     (forall (?p (SET.COL.COPRD$diagram ?p))
         (and (= (class1 (binary-coproduct-span ?p)) (SET.COL.COPRD$class1 ?p))
              (= (class2 (binary-coproduct-span ?p)) (SET.COL.COPRD$class2 ?p))
              (= (vertex (binary-coproduct-span ?p)) SET.COL$initial)
              (= (first (binary-coproduct-span ?p))
                 (SET.COL$counique (SET.COL.COPRD$class1 ?p)))
              (= (second (binary-coproduct-span ?p))
                 (SET.COL$counique (SET.COL.COPRD$class2 ?p)))))
```

- Using this span, we can show that the notion of a coproduct could be based upon pushouts and the initial class. We do this by proving the following theorem that the pushout of this span is the binary coproduct class, and the pushout injections are the coproduct injection functions.

```
     (forall (?p (SET.COL.COPRD$diagram ?p))
         (and (= (SET.COL.COPRD$binary-coproduct ?p)
                 (pushout (binary-coproduct-span ?p)))
              (= (SET.COL.COPRD$injection1 ?p)
                 (injection1 (binary-coproduct-span ?p)))
              (= (SET.COL.COPRD$injection2 ?p)
                 (injection2 (binary-coproduct-span ?p)))))
```

- We can also prove the theorem that the coproduct copairing of a pair (of classes) is the pushout copairing of the associated span.

```
     (forall (?p (SET.COL.COPRD$diagram ?p))
         (= (SET.COL.COPRD$copairing ?p)
            (copairing (binary-coproduct-span ?p))))
```

- The pushout of the opposite of a span is isomorphic to the pushout of the span. This isomorphism is mediated by the *tau* or *twist* function (for pushouts).

```
(22) (KIF$function tau-cocone)
     (= (KIF$source tau-cocone) span)
     (= (KIF$target tau-cocone) cocone)
     (forall (?r (span ?r))
         (and (= (cocone-diagram (tau-cocone ?r)) ?r)
              (= (opvertex (tau-cocone ?r)) (pushout (opposite ?r)))
              (= (opfirst (tau-cocone ?r)) (injection2 (opposite ?r)))
              (= (opsecond (tau-cocone ?r)) (injection1 (opposite ?r)))))
```

```
(23) (KIF$function tau)
     (= (KIF$source tau) span)
     (= (KIF$target tau) SET.FTN$function)
     (forall (?r (span ?r))
         (and (= (SET.FTN$source (tau ?r)) (pushout ?r))
              (= (SET.FTN$target (tau ?r)) (pushout (opposite ?r)))
              (= (tau ?r) (mediator (tau-cocone ?r)))))
```

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

```
     (forall (?r (span ?r))
         (and (= (SET.FTN$composition (tau ?r) (tau (opposite ?r)))
                 (SET.FTN$identity (pullback ?r)))
              (= (SET.FTN$composition (tau (opposite ?r)) (tau ?r))
                 (SET.FTN$identity (pullback (opposite ?r))))))
```

The Namespace of Large Relations

This namespace represents large binary relations and endorelations. The terms introduced in this namespace are listed in Table 6.

Table 6: Relation terms introduced in the IFF Core Ontology

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

Table 7 (needs expansion) lists the correspondence between standard mathematical notation and the ontological terminology in the namespace for binary relations.

Table 7: 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
$(-)^{\propto}$ or $(-)^{\perp}$ or $(-)^{op}$	‘REL$opposite’	involution – the opposite or dual relation

Relations

REL

A class relation (Figure 10) is a special case of a KIF relation with classes for its two coordinates. For class relations both (horizontal) composition and identities are defined. Horizontal composition and identity make the collections of classes and relations into a quasi-category. In the vertical direction, there is also the notion of a relation morphism, which makes this into a quasi-double-category.

$$\begin{tikzcd} \mathit{class}_2(\boldsymbol{R}) \\ \mathit{class}_1(\boldsymbol{R}) \arrow[u, "\boldsymbol{R}"'] \end{tikzcd}$$

Figure 10: Class Relation

- Let ‘`relation`’ be the SET namespace term that denotes the *binary relation* collection. A class relation $\boldsymbol{R} = \langle \mathit{class}_1(\boldsymbol{R}), \mathit{class}_2(\boldsymbol{R}), \mathit{extent}(\boldsymbol{R}) \rangle$ consists of two component classes, $\mathit{class}_1(\boldsymbol{R})$ and $\mathit{class}_2(\boldsymbol{R})$, and an *extent* class $\mathit{extent}(\boldsymbol{R}) \subseteq \mathit{class}_1(\boldsymbol{R}) \times \mathit{class}_2(\boldsymbol{R})$. We often use the following morphism notation for binary relations: $\boldsymbol{R} : \mathit{class}_1(\boldsymbol{R}) \rightarrow \mathit{class}_2(\boldsymbol{R})$. Sometimes an alternate notation for the components is desired, *source* and *target*, that follows the morphism notation. A binary relation is determined by the triple of its first, second and extent classes.

```
(1) (KIF$collection relation)
    (KIF$subcollection relation KIF$relation)

(2) (KIF$function class1)
    (KIF$function source)
    (= source class1)
    (= (KIF$source class1) relation)
    (= (KIF$target class1) SET$class)
    (KIF$restriction class1 KIF$collection1)

(3) (KIF$function class2)
(6) (KIF$function target)
    (= target class2)
    (= (KIF$source class2) relation)
    (= (KIF$target class2) SET$class)
    (KIF$restriction class2 KIF$collection2)

(4) (KIF$function extent)
    (= (KIF$source extent) relation)
    (= (KIF$target extent) SET$class)
    (KIF$restriction extent KIF$extent)
```

- 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.

```
(5) (KIF$function first)
    (= (KIF$source first) relation)
    (= (KIF$target first) SET.FTN$function)
    (forall (?r (relation ?r))
        (and (= (SET.FTN$source (first ?r)) (extent ?r))
             (= (SET.FTN$target (first ?r)) (class1 ?r))
             (forall (?x1 ?x2 ((extent ?r) [?x1 ?x2]))
                 (= ((first ?r) [?x1 ?x2]) ?x1))))

(6) (KIF$function second)
    (= (KIF$source second) relation)
    (= (KIF$target second) SET.FTN$function)
    (forall (?r (relation ?r))
        (and (= (SET.FTN$source (second ?r)) (extent ?r))
             (= (SET.FTN$target (second ?r)) (class2 ?r))
             (forall (?x1 ?x2 ((extent ?r) [?x1 ?x2]))
                 (= ((second ?r) [?x1 ?x2]) ?x2))))
```

- There is a *subrelation* relation, which is an abridgement of the KIF subcollection relation. Subrelation restricts only the extent, whereas abridgment (below) restricts only the component classes.

```
(7) (KIF$relation subrelation)
    (= (KIF$collection1 subrelation) relation)
    (= (KIF$collection2 subrelation) relation)
```

```
(KIF$abridgment subrelation KIF$subrelation)
```

- One relation *r* is an *abridgment* of another relation *s* when the first component, and the second component classes of *r* are subclasses of the first component and the second component classes of *s*, respectively, and the extent of *r* is the restriction of the extent of *s* to the component classes of *r*.

```
(8) (relation abridgment)
    (= (class1 abridgment) relation)
    (= (class2 abridgment) relation)
    (KIF$abridgment abridgment KIF$abridgment)
```

- 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.

```
(9) (KIF$function opposite)
     (= (KIF$source opposite) relation)
     (= (KIF$target opposite) 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*. The KIF function *composition* takes two composable relations and returns their composition.

```
(10) (KIF$relation composable)
     (= (KIF$collection1 composable) relation)
     (= (KIF$collection2 composable) relation)
     (forall (?r (relation ?r) ?s (relation ?s))
         (<=> (composable ?r ?s)
              (= (target ?r) (source ?s))))

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

- One can prove that the relational embedding of the composition of two functions is the relation composition of the component embeddings.

```
(forall (?f (SET.FTN$function ?f)
         ?g (SET.FTN$function ?g)
         (SET.FTN$composable ?f ?g))
    (and (composable (SET.FTN$fn2rel ?f) (SET.FTN$fn2rel ?g))
         (= (composition [(SET.FTN$fn2rel ?f) (SET.FTN$fn2rel ?g)])
            (SET.FTN$fn2rel (SET.FTN$composition [?f ?g])))))
```

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

```
(12) (KIF$function identity)
     (= (KIF$source identity) SET$class)
     (= (KIF$target identity) relation)
     (forall (?c (SET$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)))))
```

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

$$\begin{tikzcd} A \arrow[rr, "R"] \arrow[dr, "S"'] & & B \arrow[dl, "R\backslash S"] \\ & C & \end{tikzcd}$$

Figure 11: Left Residuation

$$\begin{tikzcd} & C \arrow[dl, "S/R"'] \arrow[dr, "S"] & \\ A \arrow[rr, "R"'] & & B \end{tikzcd}$$

Figure 12: Right Residuation

```
(13) (KIF$relation left-residuable)
     (= (KIF$collection1 left-residuable) relation)
     (= (KIF$collection2 left-residuable) relation)
     (forall (?r (relation ?r) ?s (relation ?s))
         (<=> (left-residuable ?r ?s)
              (= (source ?r) (source ?s))))

(14) (KIF$function left-residuation)
     (= (KIF$source left-residuation) (KIF$extent left-residuable))
     (= (KIF$target left-residuation) relation)
     (forall (?r (relation ?r) ?s (relation ?s) (left-residuable ?r ?s))
         (and (= (source (left-residuation [?r ?s])) (target ?r))
              (= (target (left-residuation [?r ?s])) (target ?s))
              (forall (?y ((target ?r) ?y) ?z ((target ?s) ?z))
                  (<=> ((left-residuation ?r ?s) ?y ?z)
                       (forall (?x ((source ?r) ?x))
                           (=> (?r ?x ?y) (?s ?x ?z)))))))

(15) (KIF$relation right-residuable)
     (= (KIF$collection1 right-residuable) relation)
     (= (KIF$collection2 right-residuable) relation)
     (forall (?r (relation ?r) ?s (relation ?s))
         (<=> (right-residuable ?r ?s)
              (= (target ?r) (target ?s))))

(16) (KIF$function right-residuation)
     (= (KIF$source right-residuation) (KIF$extent left-residuable))
     (= (KIF$target right-residuation) relation)
     (forall (?r (relation ?r) ?s (relation ?s) (right-residuable ?r ?s))
         (and (= (source (right-residuation ?r ?s)) (source ?s))
              (= (target (right-residuation ?r ?s)) (source ?r))
              (forall (?z ((source ?s) ?z) ?x ((source ?r) ?x))
                  (<=> ((right-residuation ?r ?s) ?z ?x)
                       (forall (?y ((target ?r) ?y))
                           (=> (?r ?x ?y) (?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:

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

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

$$
\begin{tikzcd}
D \arrow[r, "S"] & B \\
A \arrow[u, "T/S"] \arrow[ur, "T"] \arrow[r, "R"'] & C \arrow[u, "R\backslash T"']
\end{tikzcd}
$$

Diagram 8: Associative Law

- We can prove a general associative law (Diagram 8):

  $(R\backslash T)/S = R\backslash(T/S)$, 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])]))))
```

- Functions have a special behavior with respect to derivation.

$$
\begin{tikzcd}
A \arrow[dr, "R"'] & & B \arrow[ll, "f"'] \arrow[dl, "f^{\propto}\backslash R = f \circ R"] \\
& C &
\end{tikzcd}
\qquad
\begin{tikzcd}
& C \arrow[dl, "S/g = S \circ g^{\propto}"'] \arrow[dr, "S"] & \\
A \arrow[rr, "g"'] & & B
\end{tikzcd}
$$

Diagram 9: Pre-composition **Diagram 10: Post-composition**

  If function f and relation R are composable (Diagram 9), then

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

- If relation S and the opposite of function g are composable (Diagram 10), then

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

```
(forall (?f (SET.FTN$function ?f) ?r (relation ?r))
    (=> (= (SET.FTN$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$function ?g))
    (=> (= (target ?s) (SET.FTN$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. The KIF '`exponent`' function maps a pair of classes to its associated exponent.

```
(17) (KIF$function exponent)
     (= (KIF$source exponent) (KIF$binary-product [SET$class SET$class]))
     (= (KIF$target exponent) 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 two classes X and Y the exponent is isomorphic to the power of the product $Y^X = \wp(X \times Y)$.

```
(forall (?c1 ?c2 (SET$class ?c1) (SET$class ?c2))
   (SET.FTN$isomorphic
       (exponent [?c1 ?c2])
       (SET$power (SET.LIM.PRD$binary-product [?c1 ?c2]))))
```

- We name a special case: for any class C there is a bijective function $embed_C : \wp C \rightarrow \mathsf{REL}[1, C]$.

```
(18) (KIF$function embed)
     (= (KIF$source embed) SET$class)
     (= (KIF$target embed) 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))
                  (<=> (((embed ?c) ?b) 0 ?x)
                       (?b ?x)))))
```

- For any binary relation $\boldsymbol{R} : X_1 \rightarrow X_2$ there are fiber functions $\phi^{\boldsymbol{R}}_{12} : X_1 \rightarrow \wp X_2$ and $\phi^{\boldsymbol{R}}_{21} : X_2 \rightarrow \wp X_1$ defined as follows. Note: $\phi^{\boldsymbol{R}}_{21}$ is equivalent to the extent function of a classification.

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

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

```
(19) (KIF$function fiber12)
     (= (KIF$source fiber12) relation)
     (= (KIF$target fiber12) SET.FTN$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) ?x2))
                  (<=> (((fiber12 ?r) ?x1) ?x2)
                       (?r ?x1 ?x2)))))

(20) (KIF$function fiber21)
     (= (KIF$source fiber21) relation)
     (= (KIF$target fiber21) SET.FTN$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) ?x2))
                  (<=> (((fiber21 ?r) ?x2) ?x1)
                       (?r ?x1 ?x2)))))
```

Endorelations

REL.ENDO

- Endorelations are relations whose component classes are the same. The class *class* names this common class, and the class *extent* renames the relational extent.

```
(1) (KIF$collection endorelation)
    (KIF$subcollection endorelation relation)

(2) (KIF$function class)
    (= (KIF$source class) endorelation)
    (= (KIF$target class) SET$class)
    (forall (?r (endorelation ?r))
        (and (= (class ?r) (REL$class1 ?r))
             (= (class ?r) (REL$class2 ?r))))

(3) (KIF$function extent)
    (= (KIF$source extent) endorelation)
    (= (KIF$target extent) SET$class)
    (forall (?r (endorelation ?r))
        (= (extent ?r) (REL$extent ?r)))
```

- There is a *subendorelation* relation.

```
(4) (KIF$relation subendorelation)
    (= (KIF$collection1 subendorelation) endorelation)
    (= (KIF$collection2 subendorelation) endorelation)
    (KIF$abridgment subendorelation REL$subrelation)
```

- Two endorelations R and S are *composable* or *compatible* when the class of R is the same as the class of S. The KIF function *composition* takes two compatible endorelations and returns their composition.

```
(5) (KIF$relation composable)
    (KIF$relation compatible)
    (= composable compatible)
    (= (KIF$collection1 compatible) endorelation)
    (= (KIF$collection2 compatible) endorelation)
    (KIF$abridgment composable REL$composable)

(6) (KIF$function composition)
    (= (KIF$source composition) (KIF$extent composable))
    (= (KIF$target composition) endorelation)
    (KIF$restriction composition REL$composition)
```

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

```
(7) (KIF$function identity)
    (= (KIF$source identity) SET$class)
    (= (KIF$target identity) endorelation)
    (KIF$restriction identity REL$identity)
```

- 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.

```
(8) (KIF$function opposite)
    (= (KIF$source opposite) endorelation)
    (= (KIF$target opposite) endorelation)
    (KIF$restriction opposite REL$opposite)
```

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

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

- The KIF function *binary-intersection* takes two compatible endorelations and returns their intersection.

```
(9) (KIF$function binary-intersection)
    (= (KIF$source binary-intersection) (KIF$extent compatible))
    (= (KIF$target binary-intersection) endorelation)
    (forall (?r (endorelation ?r) ?s (endorelation ?s) (compatible ?r ?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.

```
(10) (KIF$collection reflexive)
     (KIF$subcollection 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) (KIF$collection symmetric)
     (KIF$subcollection 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.

```
(12) (KIF$collection antisymmetric)
     (KIF$subcollection 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.

```
(13) (KIF$collection transitive)
     (KIF$subcollection 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*. We use a definite description to define this.

```
(14) (KIF$function closure)
     (= (KIF$source closure) endorelation)
     (= (KIF$target closure) ORD$preorder)
     (forall (?r (endorelation ?r))
         (= (closure ?r)
            (the (?p  (ORD$preorder ?p))
                (and (subendorelation ?r ?p)
                     (forall (?o (ORD$preorder ?o))
                         (=> (subendorelation ?r ?o)
                             (subendorelation ?p ?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* – the closure of the symmetrization of *R*. We use a definite description to define this.

```
(15) (KIF$collection equivalence-relation)
     (KIF$subcollection equivalence-relation endorelation)
     (forall (?e (endorelation ?e))
         (<=> (equivalence-relation ?e)
              (and (reflexive ?e) (symmetric ?e) (transitive ?e))))

(16) (KIF$function equivalence-class)
     (= (KIF$source equivalence-class) equivalence-relation)
     (= (KIF$target equivalence-class) SET.FTN$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)
```

```
                    (?e ?x1 ?x2)))))

(17) (KIF$function quotient)
     (= (KIF$source quotient) equivalence-relation)
     (= (KIF$target quotient) SET$class)
     (forall (?e (equivalence-relation ?e))
         (= (quotient ?e)
            (SET.FTN$image (equivalence-class ?e))))

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

(19) (KIF$function equivalence-closure)
     (= (KIF$source equivalence-closure) endorelation)
     (= (KIF$target equivalence-closure) 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)))))))
```