The IFF Core Ontology

Terms in the IFF Foundation Ontology represent concepts. There are ~50 (non-parameterized) concepts in the upper metalevel corresponding to terms for collections (Figure 1). These are connected by ~850 functions representing, either attributes of these concepts, other parameterized concepts, transformations between concept, etc. There are 485 non-identical terms in the IFF Core Ontology (525 terms with 40 synonyms). These terms, which are listed in the tables in the four top-level namespaces, are partitioned into:

- 49 terms in the namespace for classes and functions;
- 131 terms in the namespace for limits in the quasi-category of classes and functions;
- 97 terms in the namespace for colimits in the quasi-category of classes and functions;
- 46 terms in the namespace for large relations;
- 84 terms in the namespace for large orders and their various morphisms, and
- 78 terms in the namespace for large graphs and their morphisms.

Terms (concepts) can be either abstract or concrete. Abstract terms are axiomatized by using concepts (hence, terms) from the ambient level upwards, whereas concrete terms also make use of terms below the ambient level. Fro example, composition in categories is abstract, whereas smallness property of categories is concrete.

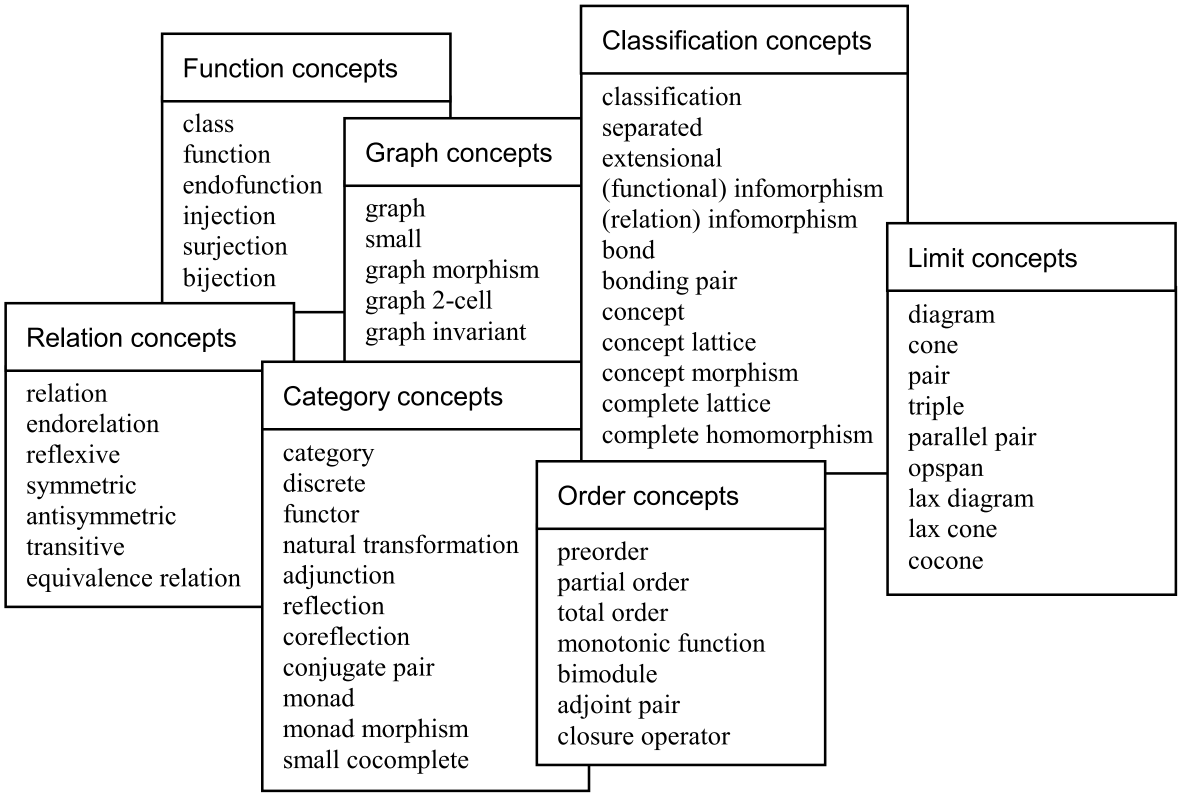


Figure 1: Upper Metalevel Concepts

The Namespace of Classes (Large Collections)

This is the main namespace in the Core (sub)Ontology of the IFF Foundation Ontology. This namespace represents classes (large collections) 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	Function	Other

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

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

Limit terminology is listed in Table 2.

Table 2: Limit terms introduced in the IFF Core Ontology

	Collection	Function	Other
SET .LIM	cone	cone-diagram = base vertex component base-shape cone-fiber limiting-cone limit projection mediator tupling-cone tupling unique	terminal = unit
SET .LIM .PRD2	diagram = pair cone	class1 class2 opposite cone-diagram vertex first second limiting-cone limit = binary-product projection1 projection2 mediator pairing tau-cone tau	
SET .LIM .PRD3	diagram = triple cone	class1 class2 class3 cone-diagram vertex first second third limiting-cone limit = ternary-product projection1 projection2 projection3 mediator tripling	
SET .LIM .EQU	diagram = parallel- pair cone	source target function1 function2 lax-parallel-pair cone-diagram vertex function limiting-cone limit = equalizer inclusion mediator kernel-diagram kernel	
SET .LIM .PBK	diagram = opspan cone	class1 class2 opvertex opfirst opsecond pair relation opposite equalizer-diagram = parallel-pair cone-diagram vertex first second limiting-cone limit = pullback projection1 projection2 mediator tripling pairing binary-product-opspan tau-cone tau kernel-pair-diagram kernel-pair	subopspan
		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- parallel-pair lax-cone	order source function1 function2 parallel-pair lax-cone-diagram vertex function limiting-lax-cone lax-limit = subequalizer subinclusion mediator	

Colimit terminology is listed in Table 3.

Table 3: Colimits terms introduced in the IFF Core Ontology

	Collection	Function	Other
SET .COL	cocone	cocone-diagram = base opvertex component base-shape cocone-fiber colimiting-cocone colimit injection comediator cotupling-cocone cotupling counique	initial = null
SET .COL .COPRD2	diagram = pair cocone	class1 class2 opposite cocone-diagram opvertex opfirst opsecond colimiting-cocone colimit = binary-coproduct injection1 injection2 comediator copairing tau-cone tau	
SET .COL .COPRD3	diagram = tri- ple cocone	class1 class2 class3 cocone-diagram opvertex opfirst opsecond opthird colimiting-cocone colimit = ternary-coproduct injection1 injection2 injection3 comediator cotripling	
SET .COL .COEQ	diagram = parallel-pair cocone	source target function1 function2 endorelation cocone-diagram opvertex function colimiting-cocone colimit = coequalizer canon comediator	
SET .COL .PSH	diagram = span cocone	class1 class2 vertex first second pair opposite coequalizer-diagram = parallel-pair cocone-diagram opvertex opfirst opsecond colimiting-cocone colimit = pushout injection1 injection2 comediator cotripling copairing binary-coproduct-span 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-opspan* : *cone* $\rightarrow$ *opspan* *vertex* : *cone* $\rightarrow$ *class* *first, second, mediator* : *cone* $\rightarrow$ *function* *limiting-cone* : *opspan* $\rightarrow$ *cone*	*binary-product* : *class* $\times$ *class* $\rightarrow$ *class* *binary-product-* *opspan* : *class* $\times$ *class* $\rightarrow$ *opspan*
	power : *class* $\rightarrow$ *relation*	

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
$\varnothing$	'`SET.LIM$null`', '`SET.LIM$initial`'	the empty class – this is the initial object in the quasi-category of classes and functions
$\times$	'`SET.LIM.PRD2$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`

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.

$$\begin{tikzcd} X \arrow[r, "f"] & Y \end{tikzcd}$$

Figure 1: Class Function

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.

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

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

Diagram 1: Restriction

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 $!_C : C \rightarrow 1$ 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 $y \in D$ there is a *constant y* function from *C* to *D*.

```
(8) (KIF$function constant)
    (= (KIF$source constant) (SET.LIM.PRD2$binary-product SET$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 (?y (?d ?y))
                 (and (= (source ((constant [?c ?d]) ?y)) ?c)
                      (= (target ((constant [?c ?d]) ?y)) ?d)
                      (forall (?x (?c ?x))
                          (= (((constant [?c ?d]) ?y) ?x) ?y))))))
    (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.

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

(11) (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.

```
(12) (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))))

(13) (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 $Y \subseteq B$.

```
(14) (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 $x \in A$.

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

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

(17) (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))))))
```

o 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])))
```

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

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

o 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)))
```

o 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.

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

(20) (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)))))
```

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

```
     (= injection monomorphism)
```

o 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.

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

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

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

```
    (= surjection epimorphism)
```

o 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.

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

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

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

```
    (= bijection isomorphism)
```

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

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

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

```
(26) (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)))))))
```

o 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.

```
(27) (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))))))))
```

o 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 $X \subseteq A$.

```
(28) (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)))))
```

o Clearly, image is related to power as follows.

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

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

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

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

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

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

```
(30) (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))))))

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

```
(32) (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))))))

(33) (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))))))
```

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

```
    (34) (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 and ternary 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 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.

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

Diagram 1: 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)))
```

○ A *cone* (Diagram 2) 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.

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

Diagram 2: Cone

```
(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 3). 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.

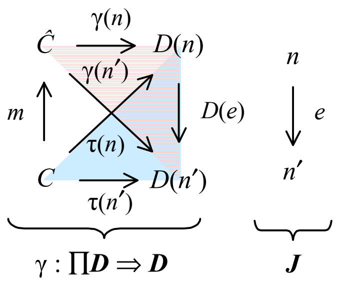

$$\begin{tikzcd} \hat{C} \arrow[r, "\gamma(n)"] \arrow[rd, "\gamma(n')" near end] & D(n) \arrow[d, "D(e)"] \\ C \arrow[u, "m"] \arrow[r, "\tau(n')"'] \arrow[ru, "\tau(n)" near start] & D(n') \end{tikzcd} \qquad \begin{tikzcd} n \arrow[d, "e"] \\ n' \end{tikzcd}$$
$\gamma : \prod \boldsymbol{D} \Rightarrow \boldsymbol{D}$ $\boldsymbol{J}$

Diagram 3: 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 convenience term ‘`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))
```

$$\underbrace{\qquad}_{\varnothing} \qquad \overset{1}{\underbrace{\qquad\qquad}_{\gamma = 1 : \prod\varnothing \Rightarrow \varnothing = 1}}$$

Figure 1: Terminal Class

- 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)))))
```

o 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.PRD2`

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

o 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)))
```

o 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]))
```

o 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))))
```

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

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

o 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)))
```

o 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))))
```

o 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" pairing. 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 [KIF$two SET.FTN$function]))
              (= (KIF$target (pairing ?d)) SET.FTN$function)
              (forall (?f1 ?f2 ((KIF$power [KIF$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)))))
```

Ternary Products

`SET.LIM.PRD3`

A *ternary product* (Figure 3) is a finite limit for a diagram of shape *three* = • • •. Such a diagram (of classes and functions) is called a *triple* of classes.

$$\begin{tikzcd} & C_1\times C_2\times C_3 \arrow[dl, "\pi_1"'] \arrow[d, "\pi_2"'] \arrow[dr, "\pi_3"] & \\ C_1 & C_2 & C_3 \end{tikzcd}$$

Figure 3: Ternary Product

- A *triple* (of classes) is the appropriate base diagram for a ternary product. Each triple consists of a triple of classes called *class1*, *class2* and *class3*. We use either the generic term '`diagram`' or the specific term '`triple`' to denote the *triple* collection. A triple is the special case of a general diagram of shape *three*.

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

(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$function class3)
    (= (KIF$source class3) diagram)
    (= (KIF$target class3) SET$class)
    (forall (?d (diagram ?d))
        (= (class3 ?d) ((GPH$class ?d) 3)))
```

- A *ternary product cone* consists of a triple of functions called *first*, *second* and *third*. These are required to have a common source class called the *vertex* of the cone. Each ternary product cone is situated over a ternary product diagram (triple). A ternary product cone is the special case of a general cone over a ternary product diagram (triple of classes).

```
(5) (KIF$collection cone)
```

```
        (KIF$subcollection cone SET.LIM$cone)
        (= cone (SET.LIM$cone-fiber GPH$three))

    (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)))

    (10) (KIF$function third)
         (= (KIF$source third) cone)
         (= (KIF$target third) SET.FTN$function)
         (forall (?r (cone ?r))
             (= (third ?r) ((SET.LIM$component ?r) 3)))
```

- The KIF function '`limiting-cone`' maps a triple of classes to its ternary product (limiting ternary product cone) (Figure 3). A limiting ternary product cone is the special case of a general limiting cone over a ternary product diagram (triple of classes).

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

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

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

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

    (15) (KIF$function projection3)
         (= (KIF$source projection3) diagram)
         (= (KIF$target projection3) SET.FTN$function)
         (forall (?d (GPH$diagram ?d)
             (= (projection3 ?d) (third (limiting-cone ?d))))
```

- There is a *mediator* function from the vertex of a ternary product cone over a ternary product diagram (triple of classes) to the ternary product of the triple. This is the unique function that commutes with the component functions of the cone. We have also introduced a "convenience term" tripling. With a diagram parameter, this maps a triple of class functions, which form a ternary product cone with the diagram, to their mediator (or *tripling*) function.

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

(17) (KIF$function tripling)
     (= (KIF$source tripling) diagram)
     (= (KIF$target tripling) KIF$partial-function)
     (forall (?d (diagram ?d))
         (and (= (KIF$source (tripling ?d))
                 (KIF$power [KIF$three SET.FTN$function]))
              (= (KIF$target (tripling ?d)) SET.FTN$function)
              (forall (?f1 ?f2 ?f3
                      ((KIF$power [KIF$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) (class3 ?d)))))))
     (KIF$restriction tripling SET.LIM$tupling)
```

- The ternary product is isomorphic to a repeated binary product in two ways: $C_1 \times C_2 \times C_3 \cong (C_1 \times C_2) \times C_3$ and $C_1 \times C_2 \times C_3 \cong C_1 \times (C_2 \times C_3)$.

```
     (forall (?c1 (SET$class c1) ?c2 (SET$class ?c2) ?c3 (SET$class ?c3)
              ?p12 (SET.LIM.PRD2$pair ?p12) ?p23 (SET.LIM.PRD2$pair ?p23)
              ?p12-3 (SET.LIM.PRD2$pair ?p12-3) ?p1-23 (SET.LIM.PRD2$pair ?p1-23)
              ?t (triple ?t))
         (=> (and (= (SET.LIM.PRD2$class1 ?p12) ?c1)
                  (= (SET.LIM.PRD2$class2 ?p12) ?c2)
                  (= (SET.LIM.PRD2$class1 ?p23) ?c2)
                  (= (SET.LIM.PRD2$class2 ?p23) ?c3)
                  (= (SET.LIM.PRD2$class1 ?p12-3) (SET.LIM.PRD2$binary-product ?p12))
                  (= (SET.LIM.PRD2$class2 ?p12-3) ?c3)
                  (= (SET.LIM.PRD2$class1 ?p1-23) ?c1)
                  (= (SET.LIM.PRD2$class2 ?p1-23) (SET.LIM.PRD2$binary-product ?p23))
                  (= (class1 ?t) ?c1) (= (class2 ?t) ?c2) (= (class3 ?t) ?c3))
             (and (SET.FTN$isomorphic
                      (ternary-product ?t)
                      (SET.LIM.PRD2$binary-product-product ?p12-3))
                  (SET.FTN$isomorphic
                      (ternary-product ?t)
                      (SET.LIM.PRD2$binary-product-product ?p1-23)))))
```

Equalizers

SET.LIM.EQU

An *equalizer* (Figure 4) 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 4: 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)))
```

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

```
(6) (KIF$function lax-parallel-pair)
    (= (KIF$source lax-parallel-pair) diagram)
    (= (KIF$target lax-parallel-pair) SET.LIM.SEQU$lax-diagram)
    (forall (?p (diagram ?p))
        (and (= (SET.LIM.SEQU$order (lax-parallel-pair ?p)) (ORD$identity (target ?p)))
             (= (SET.LIM.SEQU$source (lax-parallel-pair ?p)) (source ?p))
             (= (SET.LIM.SEQU$function1 (lax-parallel-pair ?p)) (function1 ?p))
             (= (SET.LIM.SEQU$function2 (lax-parallel-pair ?p)) (function2 ?p))))
```

- 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.

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

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

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

(10) (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 4). A limiting equalizer cone is the special case of a general limiting cone over an equalizer diagram (parallel pair of functions).

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

```
(12) (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))))

(13) (KIF$function inclusion)
     (= (KIF$source inclusion) diagram)
     (= (KIF$target inclusion) SET.FTN$function)
     (forall (?d (diagram ?d))
         (= (inclusion ?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.

```
(14) (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.

```
(15) (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)))

(16) (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`

A *pullback* (Figure 5) is a finite limit for a diagram of shape *opspan* = • → • ← •. Such a diagram (of classes and functions) is called an *opspan*.

$$\begin{tikzcd}
& C_1{\times_{C_3}}C_2 \arrow[dl, "\pi_1"'] \arrow[dr, "\pi_2"] & \\
C_1 \arrow[dr, "f_1"'] & & C_2 \arrow[dl, "f_2"] \\
& C_3 &
\end{tikzcd}$$

Figure 5: Pullback

- 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)))
```

o 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)))))
```

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

```
    (8) (KIF$function pair)
        (= (KIF$source pair) diagram)
        (= (KIF$target pair) SET.LIM.PRD2$diagram)
        (= pair
           (GPH$restriction [GPH$two GPH$opspan]))
```

o 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))))))
```

o 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))))
```

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

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

- The *parallel pair* or *equalizer diagram* function maps an opspan of functions to the associated (`SET.LIM.EQU`) parallel pair (see Figure 6) of functions, which are the composite of the product projections of the binary product of the pair of classes overlying the opspan with the opfirst and opsecond functions of the opspan. The equalizer and inclusion of the parallel pair can be used to define the pullback and pullback projections.

$$\begin{tikzcd}
C_1 \arrow[dr, "f_1"'] & C_1\times C_2 \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] \arrow[d, shift left] \arrow[d, shift right] & C_2 \arrow[dl, "f_2"] \\
& B &
\end{tikzcd}$$

Figure 6: Equalizer Diagram

```
(11) (KIF$function equalizer-diagram)
     (KIF$function parallel-pair)
     (= parallel-pair equalizer-diagram)
     (= (KIF$source equalizer-diagram) diagram)
     (= (KIF$target equalizer-diagram) SET.LIM.EQU$diagram)
     (forall (?s (diagram ?s))
         (and (= (SET.LIM.EQU$target (equalizer-diagram ?s))
                 (opvertex ?s))
              (= (SET.LIM.EQU$source (equalizer-diagram ?s))
                 (SET.LIM.PRD2$binary-product (pair ?s)))
              (= (SET.LIM.EQU$function1 (equalizer-diagram ?s))
                 (SET.FTN$composition
                     [(SET.LIM.PRD2$projection1 (pair ?s))
                      (opfirst ?s)]))
              (= (SET.LIM.EQU$function2 (equalizer-diagram ?s))
                 (SET.FTN$composition
                     [(SET.LIM.PRD2$projection2 (pair ?s))
                      (opsecond ?r)]))))
```

- 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).

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

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

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

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

(16) (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 5). A limiting pullback cone is the special case of a general limiting cone over a pullback diagram (opspan). The last axiom expresses concreteness of the limit – it expresses pullbacks in terms of products and equalizers: the limit of the equalizer diagram is (not just isomorphic but) equal to the pullback; likewise, the compositions of the inclusion of the equalizer diagram with the product projections of the pair diagram are equal to the pullback projections.

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

(18) (KIF$function limit)
     (KIF$function pullback)
     (= pullback limit)
     (= (KIF$source limit) diagram)
     (= (KIF$target limit) SET$class)

(19) (KIF$function projection1)
     (= (KIF$source projection1) diagram)
     (= (KIF$target projection1) SET.FTN$function)

(20) (KIF$function projection2)
     (= (KIF$source projection2) diagram)
     (= (KIF$target projection2) SET.FTN$function)

(21) (forall (?d (diagram ?d))
         (and (= (limit ?d) (vertex (limiting-cone ?d)))
              (= (projection1 ?d) (first (limiting-cone ?d)))
              (= (projection2 ?d) (second (limiting-cone ?d)))))

(22) (forall (?d (diagram ?d))
         (and (= (limit ?d)
                 (SET.LIM.EQU$limit (equalizer-diagram ?d)))
              (= (projection1 ?d)
                 (SET.FTN$composition
                     [(SET.LIM.EQU$inclusion (equalizer-diagram ?d))
                      (SET.LIM.PRD2$projection1 (pair ?d))]))
              (= (projection2 ?d)
                 (SET.FTN$composition
                     [(SET.LIM.EQU$inclusion (equalizer-diagram ?d))
                      (SET.LIM.PRD2$projection2 (pair ?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 convenience term '`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.

```
(23) (KIF$function mediator)
     (= (KIF$source mediator) cone)
     (= (KIF$target mediator) SET.FTN$function)
```

```
          (KIF$restriction mediator SET.LIM$mediator)

     (24) (KIF$partial-function tripling)
          (= (KIF$source tripling) diagram)
          (= (KIF$target tripling) KIF$partial-function)
          (forall (?d (diagram ?d))
              (and (= (KIF$source (tripling ?d))
                      (KIF$power [KIF$three SET.FTN$function]))
                   (= (KIF$target (tripling ?d)) SET.FTN$function)
                   (forall (?f1 ?f2 ?f3
                            ((KIF$power [KIF$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)

     (25) (KIF$function pairing)
          (= (KIF$source pairing) diagram)
          (= (KIF$target pairing) KIF$partial-function)
          (forall (?d (diagram ?d))
              (and (= (KIF$source (pairing ?d)) (KIF$power [KIF$two SET.FTN$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.

```
     (26) (KIF$function binary-product-opspan)
          (= (KIF$source binary-product-opspan) SET.LIM.PRD2$diagram)
          (= (KIF$target binary-product-opspan) diagram)
          (forall (?p (SET.LIM.PRD2$diagram ?p))
              (and (= (class1 (binary-product-opspan ?p)) (SET.LIM.PRD2$class1 ?p))
                   (= (class2 (binary-product-opspan ?p)) (SET.LIM.PRD2$class2 ?p))
                   (= (opvertex (binary-product-opspan ?p)) SET.LIM$terminal)
                   (= (opfirst (binary-product-opspan ?p))
                      (SET.LIM$unique (SET.LIM.PRD2$class1 ?p)))
                   (= (opsecond (binary-product-opspan ?p))
                      (SET.LIM$unique (SET.LIM.PRD2$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.PRD2$binary-product ?p)
                      (pullback (binary-product-opspan ?p)))
                   (= (SET.LIM.PRD2$projection1 ?p)
                      (projection1 (binary-product-opspan ?p)))
                   (= (SET.LIM.PRD2$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.PRD2$diagram ?p))
            (= (SET.LIM.PRD2$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).

```
(27) (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)))))

(28) (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 $f : A \rightarrow B$ there is a *kernel-pair* equivalence relation on the source set A.

```
(29) (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)))

(30) (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 convenience terms.

- 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 7)

  – 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

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

  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 $f_2 \cdot \phi^S_1 = \phi^S_{21}$.

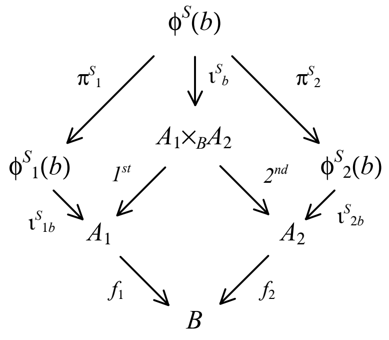

$$\begin{tikzcd}
& & \phi^S(b) \arrow[dll, "\pi^S_1"'] \arrow[d, "\iota^S_b"] \arrow[drr, "\pi^S_2"] & & \\
\phi^S_1(b) \arrow[dr, "\iota^S_{1b}"'] & & A_1{\times_B}A_2 \arrow[dl, "1^{st}"'] \arrow[dr, "2^{nd}"] & & \phi^S_2(b) \arrow[dl, "\iota^S_{2b}"] \\
& A_1 \arrow[dr, "f_1"'] & & A_2 \arrow[dl,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

Figure 7: Pullback Fibers

```
(31) (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))))))

(32) (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)))))

(33) (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)))))

     (34) (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)))))

     (35) (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)))))

     (36) (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))))))

     (37) (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))))))

     (38) (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))))))

     (39) (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]))))))

    (40) (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]))))))

    (41) (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))))))

    (42) (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))))))
```

- Pullback fibers are the same as the fibers of the relation of a pullback diagram (opspan). However, since the relational fiber embeddings are into the pullback class itself, they are a composite of the pullback fiber embeddings.

```
         (forall (?s (opspan ?s))
             (and (= (fiber12 (relation ?s))
                     (fiber12 ?s))
                  (= (fiber21 (relation ?s))
                     (fiber21 ?s))))

         (forall (?s (opspan ?s))
             (and (forall (?x1 ((class1 ?s) ?x1))
                      (= ((fiber12-embedding (relation ?s)) ?x1)
                         (SET.FTN$composition
                             [((fiber12-embedding ?s) ?x1)
                              ((fiber-embedding ?s) ((opfirst ?s) ?x1))])))
                  (forall (?x2 ((class2 ?s) ?x2))
                      (= ((fiber21-embedding (relation ?s)) ?x2)
                         (SET.FTN$composition
                             [((fiber21-embedding ?s) ?x2)
                              ((fiber-embedding ?s) ((opsecond ?s) ?x2))])))))
```

Subequalizers

SET.LIM.SEQU

A *subequalizer* (Figure 8) 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 8: Subequalizer

- A *lax parallel pair* $f_1, f_2 : A \rightarrow \boldsymbol{B} = \langle B, \leq \rangle$ 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)))))
```

- 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.

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

```
(7) (KIF$collection lax-cone)
```

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

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

(10) (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 8). 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 ‘`subinclusion`’, 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.

```
(11) (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))

(12) (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))))

(13) (KIF$function subinclusion)
     (= (KIF$source subinclusion) lax-diagram)
     (= (KIF$target subinclusion) SET.FTN$function)
     (forall (?d (lax-diagram ?d))
        (= (subinclusion ?d) (function (limiting-lax-cone ?d))))

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

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

```
(14) (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 (subinclusion (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 <u>specific</u> 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.

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

Diagram 1: Diagrams, Co-cones and Fibers

- A *cocone* (Diagram 2) 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.

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

Diagram 2: Cocone

```
(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)

(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* 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 3). 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.

$$\begin{tikzcd}
n \arrow[d, "e"'] & D(n) \arrow[r, "\gamma(n)"] \arrow[d, "D(e)"'] \arrow[dr, "\tau(n)"] & \hat{C} \arrow[d, "m"] \\
n' & D(n') \arrow[r, "\tau(n')"'] \arrow[ur, "\gamma(n')"] & C
\end{tikzcd}$$

$\boldsymbol{J} \qquad \gamma : \boldsymbol{D} \Rightarrow \sum\boldsymbol{D}$

Diagram 3: 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 convenience term '`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))
```

$$\underbrace{\qquad}_{\varnothing} \qquad \overset{0}{\underbrace{\qquad\qquad}}_{\gamma = 0 : \varnothing \Rightarrow \Sigma\varnothing = 0}$$

Figure 1: Initial Class

- 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.COPRD2

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

Figure 2: Binary Coproduct

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

- A *pair* (of classes) is the appropriate base diagram for a binary 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 2). 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" <u>copairing</u>. 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 [KIF$two SET.FTN$function]))
                  (= (KIF$target (copairing ?d)) SET.FTN$function)
                  (forall (?f1 ?f2 ((KIF$power [KIF$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))))))
```

Ternary Coproducts

`SET.COL.COPRD3`

A *ternary coproduct* (Figure 3) is a finite colimit for a diagram of shape *three* = • • •. Such a diagram (of classes and functions) is called a *triple* of classes.

$$\begin{tikzcd}
C_1 \arrow[dr, "\iota_1"'] & C_2 \arrow[d, "\iota_2"] & C_3 \arrow[dl, "\iota_3"] \\
 & C_1+C_2+C_3 &
\end{tikzcd}$$

Figure 3: Ternary Coproduct

- A *triple* (of classes) is the appropriate base diagram for a ternary coproduct. Each triple consists of a triple of classes called *class1*, *class2* and *class3*. We use either the generic term '`diagram`' or the specific term '`triple`' to denote the *triple* collection. A triple is the special case of a general diagram of shape *three*.

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

   (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$function class3)
       (= (KIF$source class3) diagram)
       (= (KIF$target class3) SET$class)
       (forall (?d (diagram ?d))
           (= (class3 ?d) ((GPH$class ?d) 3)))
```

- A *ternary coproduct cocone* consists of a triple of functions called *opfirst*, *opsecond* and *opthird*. These are required to have a common target class called the *opvertex* of the cocone. Each ternary coproduct cocone is situated under a ternary coproduct diagram (triple). A ternary coproduct cocone is the special case of a general cocone under a ternary coproduct diagram (triple of classes).

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

   (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)))

(10) (KIF$function opthird)
     (= (KIF$source opthird) cocone)
     (= (KIF$target opthird) SET.FTN$function)
     (forall (?s (cocone ?s))
         (= (opthird ?s) ((SET.COL$component ?s) 3)))
```

- The KIF function ‘`colimiting-cocone`’ maps a triple of classes to its ternary coproduct (colimiting ternary coproduct cocone) (Figure 3). A colimiting ternary coproduct cocone is the special case of a general colimiting cocone over a ternary coproduct diagram (triple of classes).

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

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

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

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

(15) (KIF$function injection3)
     (= (KIF$source injection3) diagram)
     (= (KIF$target injection3) SET.FTN$function)
     (forall (?d (GPH$diagram ?d)
         (= (injection3 ?d) (opthird (colimiting-cocone ?d))))
```

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

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

(17) (KIF$function cotripling)
     (= (KIF$source cotripling) diagram)
     (= (KIF$target cotripling) KIF$partial-function)
     (forall (?d (diagram ?d))
         (and (= (KIF$source (cotripling ?d))
                 (KIF$power [KIF$three SET.FTN$function]))
              (= (KIF$target (cotripling ?d)) SET.FTN$function)
              (forall (?f1 ?f2 ?f3
                     ((KIF$power [KIF$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) (class3 ?d)))))))
     (KIF$restriction cotripling SET.COL$cotupling)
```

Coequalizers

`SET.COL.COEQ`

A *coequalizer* (Figure 4) 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 4: 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)))
```

- The information in a coequalizer diagram (parallel pair of functions) is equivalently expressed as an *endorelation* on the target class.

```
(6) (KIF$function endorelation)
```

```
    (= (KIF$source endorelation) diagram)
    (= (KIF$target endorelation) REL.ENDO$endorelation)
    (forall (?d (diagram ?d))
        (and (= (REL.ENDO$class (endorelation ?d)) (target ?d))
             (forall (?y1 ((target ?d) ?y1) ?y2 ((target ?d) ?y2))
                 (<=> ((endorelation ?d) ?y1 ?y2)
                      (exists (?x ((source ?d) ?x))
                          (and (= ?y1 ((function1 ?d) ?x))
                               (= ?y2 ((function2 ?d) ?x))))))))
```

- A *coequalizer cocone* consists of an *opvertex* class and a *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.

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

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

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

(10) (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 4). A colimiting *coequalizer cocone* is the special case of a general colimiting cocone over a coequalizer diagram (parallel pair of functions). The *coequalizer* and *canon* are expressed both as restrictions of generic colimits and, in the last axiom, as the quotient and canon of the endorelation of the diagram.

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

(12) (KIF$function colimit)
     (KIF$function coequalizer)
     (= coequalizer colimit)
     (= (KIF$source colimit) diagram)
     (= (KIF$target colimit) SET$class)

(13) (KIF$function canon)
     (= (KIF$source canon) diagram)
     (= (KIF$target canon) SET.FTN$function)

(14) (forall (?d (diagram ?d))
         (and (= (colimit ?d) (opvertex (colimiting-cocone ?d)))
              (= (canon ?d) (function (colimiting-cocone ?p)))))

(15) (forall (?d (diagram ?d))
         (and (= (colimit ?d)
                 (REL.ENDO$quotient
                     (REL.ENDO$equivalence-closure (endorelation ?d))))
              (= (canon ?d)
                 (REL.ENDO$canon
                     (REL.ENDO$equivalence-closure (endorelation ?d))))))
```

- 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.

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

Pushouts

`SET.COL.PSH`

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

Figure 5: Pushout

A *pushout* (Figure 5) is a finite colimit for a diagram of shape *span* = $\bullet \leftarrow \bullet \rightarrow \bullet$. A diagram of this shape is called a *span* of classes and functions.

- 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$target pair) SET.COL.COPRD2$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))
```

- The *parallel pair* or *coequalizer diagram* function maps a span to the associated (SET.COL.COEQ) parallel pair (see Figure 6) of functions, which are the composite of the first and second functions of the span with the coproduct injections of the binary coproduct of the pair of classes underlying the span. The coequalizer and canon of the parallel pair can be used to define the pushout and pushout injections.

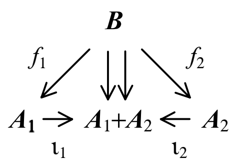

$$\begin{tikzcd} & B \arrow[dl, "f_1"'] \arrow[d, shift left] \arrow[d, shift right] \arrow[dr, "f_2"] & \\ A_1 \arrow[r, "\iota_1"'] & A_1+A_2 & A_2 \arrow[l, "\iota_2"] \end{tikzcd}$$

Figure 6: Coequalizer Diagram

```
    (9) (KIF$function coequalizer-diagram)
        (KIF$function parallel-pair)
        (= parallel-pair coequalizer-diagram)
        (= (KIF$source coequalizer-diagram) diagram)
        (= (KIF$target coequalizer-diagram) SET.COL.COEQ$diagram)
        (forall (?r (diagram ?r))
            (and (= (SET.COL.COEQ$source (coequalizer-diagram ?r))
                    (vertex ?r))
                 (= (SET.COL.COEQ$target (coequalizer-diagram ?r))
                    (SET.COL.COPRD2$binary-coproduct (pair ?r)))
                 (= (SET.COL.COEQ$function1 (coequalizer-diagram ?r))
                    (SET.FTN$composition
                        [(first ?r)
                         (SET.COL.COPRD2$injection1 (pair ?r))]))
                 (= (SET.COL.COEQ$function2 (coequalizer-diagram ?r))
                    (SET.INFO$composition
                        [(second ?r)
                         (SET.COL.COPRD2$injection2 (pair ?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).

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

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

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

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

    (14) (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 a span to its pushout (colimiting pushout cocone) (Figure 5). A colimiting pushout cocone is the special case of a general colimiting cocone over a pushout diagram (span). The last axiom expresses concreteness of the colimit – it expresses pushouts in terms of coproducts and coequalizers: the colimit of the coequalizer diagram is (not just isomorphic but) equal to the pushout; likewise, the compositions of the coproduct injections of the pair diagram with the canon of the coequalizer diagram are equal to the pushout injections.

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

(16) (KIF$function colimit)
     (KIF$function pushout)
     (= pushout colimit)
     (= (KIF$source colimit) diagram)
     (= (KIF$target colimit) SET$class)

(17) (KIF$function injection1)
     (= (KIF$source injection1) diagram)
     (= (KIF$target injection1) SET.FTN$function)

(18) (KIF$function injection2)
     (= (KIF$source injection2) diagram)
     (= (KIF$target injection2) SET.FTN$function)

(19) (forall (?d (diagram ?d))
         (and (= (colimit ?d) (opvertex (colimiting-cocone ?d)))
              (= (injection1 ?d) (opfirst (colimiting-cocone ?d)))
              (= (injection2 ?d) (opsecond (colimiting-cocone ?d)))))

(20) (forall (?d (diagram ?d))
         (and (= (colimit ?d)
                 (SET.COL.COEQ$colimit (coequalizer-diagram ?d)))
              (= (injection1 ?d)
                 (SET.FTN$composition
                     [(SET.COL.COPRD2$injection1 (pair ?d))
                      (SET.COL.COEQ$canon (coequalizer-diagram ?d))]))
              (= (injection2 ?d)
                 (SET.FTN$composition
                     [(SET.COL.COPRD2$injection2 (pair ?d))
                      (SET.COL.COEQ$canon (coequalizer-diagram ?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 convenience term ‘`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.

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

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

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

```
(22) (KIF$function binary-coproduct-span)
     (= (KIF$source binary-coproduct-span) SET.COL.COPRD2$diagram)
     (= (KIF$target binary-coproduct-span) diagram)
     (forall (?p (SET.COL.COPRD2$diagram ?p))
         (and (= (class1 (binary-coproduct-span ?p)) (SET.COL.COPRD2$class1 ?p))
              (= (class2 (binary-coproduct-span ?p)) (SET.COL.COPRD2$class2 ?p))
              (= (vertex (binary-coproduct-span ?p)) SET.COL$initial)
              (= (first (binary-coproduct-span ?p))
                 (SET.COL$counique (SET.COL.COPRD2$class1 ?p)))
              (= (second (binary-coproduct-span ?p))
                 (SET.COL$counique (SET.COL.COPRD2$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.COPRD2$diagram ?p))
         (and (= (SET.COL.COPRD2$binary-coproduct ?p)
                 (pushout (binary-coproduct-span ?p)))
              (= (SET.COL.COPRD2$injection1 ?p)
                 (injection1 (binary-coproduct-span ?p)))
              (= (SET.COL.COPRD2$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.COPRD2$diagram ?p))
         (= (SET.COL.COPRD2$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).

```
(23) (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)))))

(24) (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 1.

Table 1: Relation terms introduced in the IFF Core Ontology

	Collection	Function	Other

`REL`	`relation`	`class1 class2 extent first second`
		`opposite composition identity`
		`left-residuation right-residuation`
		`exponent embed`
		`fiber12 fiber21`
		`fiber12-embedding fiber21-embedding`
		`empty`
			`subrelation`
			`abridgment`
			`composable-opspan`
			`composable`
			`left-residuable-opspan`
			`left-residuable`
			`right-residuable-opspan`
			`right-residuable`

`REL`
`.ENDO`
	`endorelation`
	`reflexive`
	`symmetric`
	`antisymmetric`
	`transitive`
	`equivalence-`
	`relation`
		`class extent`
		`opposite composition identity`
		`binary-intersection`
		`closure`
		`equivalence-class quotient canon`
		`equivalence-closure`
			`subendorelation`
			`composable`
			`three`

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

Table 2: Correspondence between Mathematical Notation and Ontological Terminology

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

Relations

REL

A class relation (Figure 1) 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 1: 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$opspan composable-opspan)
         (= composable-opspan [target source])

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

    (12) (KIF$function composition)
         (= (KIF$source composition) (KIF$pullback composable-opspan))
         (= (KIF$target composition) relation)
         (forall (?r1 (relation ?r1) ?r2 (relation ?r2) (composable ?r1 ?r2))
             (and (= (source (composition [?r1 ?r2])) (source ?r1))
                  (= (target (composition [?r1 ?r2])) (target ?r2))
                  (forall (?x ((source ?r1) ?x) ?z ((target ?r2) ?z))
                      (<=> ((composition [?r1 ?r2]) ?x ?z)
                           (exists (?y ((target ?r1) ?y))
                               (and (?r1 ?x ?y) (?r2 ?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 $identity_A$.

```
    (13) (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))
                      (<=> ((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 residuation* (Figure 2) that takes two left residuable relations and returns their left residuation. Dually,

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

Figure 2: Left Residuation

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

Figure 3: Right Residuation

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 residuation* (Figure 3) that takes two right residuable relations and returns their right residuation.

```
(14) (KIF$opspan left-residuable-opspan)
     (= left-residuable-opspan [source source])

(15) (KIF$relation left-residuable)
     (= (KIF$collection1 left-residuable) relation)
     (= (KIF$collection2 left-residuable) relation)
     (= (KIF$extent left-residuable) (KIF$pullback left-residuable-opspan))

(16) (KIF$function left-residuation)
     (= (KIF$source left-residuation) (KIF$pullback left-residuable-opspan))
     (= (KIF$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)))))))

(17) (KIF$opspan right-residuable-opspan)
     (= right-residuable-opspan [target target])

(18) (KIF$relation right-residuable)
     (= (KIF$collection1 right-residuable) relation)
     (= (KIF$collection2 right-residuable) relation)
     (= (KIF$extent right-residuable) (KIF$pullback right-residuable-opspan))

(19) (KIF$function right-residuation)
     (= (KIF$source right-residuation) (KIF$pullback right-residuable-opspan))
     (= (KIF$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])))))
```

o 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)))
```

o 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

o 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])]))))
```

o Functions have a special behavior with respect to derivation.

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

Diagram 4: Pre-composition

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

Diagram 5: Post-composition

If function f and relation R are composable (Diagram 4), then

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

- If relation S and the opposite of function g are composable (Diagram 5), 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.

```
(20) (KIF$function exponent)
     (= (KIF$source exponent) (KIF$binary-product [SET$class SET$class]))
     (= (KIF$target exponent) SET$class)
     (forall (?c1 (SET$class ?c1) ?c2 (SET$class ?c2))
         (and (KIF$subcollection (exponent [?c1 ?c2]) relation)
              (forall (?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.PRD2$binary-product [?c1 ?c2]))))
```

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

```
(21) (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:

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

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

  When relations are interpreted as classifications, the fiber $\phi^{\boldsymbol{R}}_{12}$ is the intent function and the fiber $\phi^{\boldsymbol{R}}_{21}$ the extent function. Associated with these are two embedding functions into the extent of the relation: $\iota^{\boldsymbol{R}}_{12a1} : \phi^{\boldsymbol{R}}_{12}(a_1) \rightarrow R$ and $\iota^{\boldsymbol{R}}_{21a2} : \phi^{\boldsymbol{R}}_{21}(a_2) \rightarrow R$.

```
(22) (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$power (class2 ?r)))
              (= (fiber12 ?r)
                 (SET.FTN$composition
                     [(SET.FTN$fiber (first ?r)) (SET.FTN$power (second ?r))]))))

(23) (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$power (class1 ?r)))
                 (= (fiber21 ?r)
                    (SET.FTN$composition
                        [(SET.FTN$fiber (second ?r)) (SET.FTN$power (first ?r))]))))

    (24) (KIF$function fiber12-embedding)
         (= (KIF$source fiber12-embedding) relation)
         (= (KIF$target fiber12-embedding) KIF$function)
         (forall (?r (relation ?r))
             (and (= (KIF$source (fiber12-embedding ?r)) (class1 ?r))
                  (= (KIF$target (fiber12-embedding ?r)) SET.FTN$function)
                  (forall (?x1 ((class1 ?r) ?x1))
                      (and (= (SET.FTN$source ((fiber12-embedding ?r) ?x1))
                              ((fiber12 ?r) ?x1))
                           (= (SET.FTN$target ((fiber12-embedding ?r) ?x1))
                              (class2 ?r))
                           (= ((fiber12-embedding ?r) ?x1)
                              (SET.FTN$inclusion
                                  [((SET.FTN$fiber (first ?r)) ?x1) (extent ?r)]))))))

    (25) (KIF$function fiber21-embedding)
         (= (KIF$source fiber21-embedding) relation)
         (= (KIF$target fiber21-embedding) KIF$function)
         (forall (?r (relation ?r))
             (and (= (KIF$source (fiber21-embedding ?r)) (class2 ?r))
                  (= (KIF$target (fiber21-embedding ?r)) SET.FTN$function)
                  (forall (?x2 ((class2 ?r) ?x2))
                      (and (= (SET.FTN$source ((fiber21-embedding ?r) ?x2))
                              ((fiber21 ?r) ?x2))
                           (= (SET.FTN$target ((fiber21-embedding ?r) ?x2))
                              (class1 ?r))
                           (= ((fiber21-embedding ?r) ?x2)
                              (SET.FTN$inclusion [((fiber21 ?r) ?x2) (class1 ?r)]))))))
                           (= ((fiber21-embedding ?r) ?x2)
                              (SET.FTN$inclusion
                                  [((SET.FTN$fiber (second ?r)) ?x2) (extent ?r)]))))))
```

o For any two classes *A* and *B* there is an empty relation $\emptyset_{A,B} \subseteq A \times B$.

```
    (26) (KIF$function empty)
         (= (KIF$source empty) (KIF$binary-product [SET$class SET$class]))
         (= (KIF$target empty) relation)
         (forall (?c1 (SET$class ?c1) ?c2 (SET$class ?c2))
             (and (= (collection1 (empty [?c1 ?c2])) ?c1)
                  (= (collection2 (empty [?c1 ?c2])) ?c2)
                  (= (extent (empty [?c1 ?c2])) SET$empty)))
```

Endorelations

REL.ENDO

o 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)))))
```

o 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)))
```

o 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)))))))
```

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)))))))
```

Examples

- The endorelation *three* underlies the ordinal *three*.

```
(20) (endorelation three)
     (= (class three) SET$three)
     (three 1 2)
     (three 2 3)
     (forall (?x (SET$three ?x) ?y (SET$three ?y))
         (<=> (three ?x ?y)
              (and (= [?x ?y] [1 2])
                   (= [?x ?y] [2 3]))))
```

The Namespace of Large Orders

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

Table 1: Terms introduced into the large order namespace

	Collection	Function	Other

`ORD`	`preorder`
	`partial-order`
	`total-order`
		`class = underlying extent`
		`opposite classification category`
		`identity power`
		`up down` (uses ∃ quantifier)
		`up-class down-class`
		`up-embedding down-embedding`
		`greatest least`
		`upper-bound lower-bound` (uses ∀ quantifier)
		`upper-lower-closure lower-upper-closure`
		`join-dense meet-dense`
			`suborder`
			`three`

`ORD .FTN`	`monotonic-function`	`source target function`
		`extent functor inclusion`
		`subextent natural-transformation`
		`opposite composition identity`
		`left right`
			`subfunction`
			`restriction`
			`composable-opspan`
			`composable`

`ORD .REL`	`bimodule`	`source target relation`
		`opposite composition identity`
			`composable-opspan`
			`composable`

`ORD .ADJ`	`adjoint-pair`	`source target left right`
		`source-closure target-closure`
		`closure-operator co-closure-operator`
		`source-closed-preorder target-closed-preorder`
		`left-closed right-closed`
		`infomorphism adjunction bond`
		`opposite composition identity`
		`closure-operator`
			`composable-opspan`
			`composable`

`ORD .CLO`	`closure-structure = closure-operator`	`preorder closure`
		`closed closed-preorder`
		`extent`
		`monad adjoint-pair`	

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

Table 2: Correspondence between Mathematical Notation and Ontological Terminology

Math	Ontological Terminology	Natural Language Description
$\leq$	`ORD$extent`	the order relation class of a preorder

Orders

ORD

$$\begin{tikzcd} \mathsf{class}(\boldsymbol{A}) \\ \mathsf{class}(\boldsymbol{A}) \arrow[u, "\boldsymbol{A}"'] \end{tikzcd}$$

Figure 1: Preorder as Classification

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

```
(1) (KIF$collection preorder)
    (KIF$subcollection preorder REL.ENDO$endorelation)

(2) (KIF$function class)
    (KIF$function underlying)
    (= underlying class)
    (= (KIF$source class) preorder)
    (= (KIF$target class) SET$class)
    (KIF$restriction class REL.ENDO$class)

(3) (KIF$function extent)
    (= (KIF$source extent) preorder)
    (= (KIF$target extent) SET$class)
    (KIF$restriction extent REL.ENDO$extent)
```

- One preorder $\boldsymbol{A}_1 = \langle A_1, \leq_1 \rangle$ is a *suborder* of another preorder $\boldsymbol{A}_2 = \langle A_2, \leq_2 \rangle$ when $A_1 \subseteq A_2$ and $x \leq_1 y$ implies $x \leq_2 y$ for all pairs of elements $x, y \in A_1$.

```
(4) (KIF$relation suborder)
    (= (KIF$collection1 suborder) preorder)
    (= (KIF$collection2 suborder) preorder)
    (forall (?o1 (preorder ?o1) ?o2 (preorder ?o2))
        (<=> (suborder ?o1 ?o2)
             (and (SET$subclass (class ?o1) (class ?o2))
                  (forall (?x ((class ?o1) ?x) ?y ((class ?o1) ?y))
                      (=> (?o1 ?x ?y) (?o2 ?x ?y))))))
```

- To each preorder $\boldsymbol{A} = \langle A, \leq_A \rangle$, there is an *opposite* preorder $\boldsymbol{A}^{\mathrm{op}} = \langle A, \leq_A^{\mathrm{op}} \rangle = \langle A, \geq_A \rangle$.

```
(5) (KIF$function opposite)
    (= (KIF$source opposite) preorder)
    (= (KIF$target opposite) preorder)
    (KIF$restriction opposite REL.ENDO$opposite)
```

- Since any preorder $\boldsymbol{A} = \langle A, \leq_A \rangle = \langle \mathsf{class}(\boldsymbol{A}), \mathsf{ext}(\boldsymbol{A}) \rangle$ is an endorelation, it is also a relation. But relations are synonymous with *classification*s. We name this manifestation of a preorder $\mathsf{cls}(\boldsymbol{A}) = \langle \mathsf{class}(\boldsymbol{A}), \mathsf{class}(\boldsymbol{A}), \mathsf{ext}(\boldsymbol{A}) \rangle$.

```
(6) (KIF$function classification)
    (= (KIF$source classification) preorder)
    (= (KIF$target classification) CLS$classification)
    (forall (?o (preorder ?o))
        (and (= (instance (classification ?o)) (class ?o))
             (= (type (classification ?o)) (class ?o))
             (= (incidence (classification ?o)) (extent ?o))))
```

- Any preorder is a *category*.

```
(7) (KIF$function category)
    (= (KIF$source category) preorder)
    (= (KIF$target category) CAT$category)
    (forall (?o (preorder ?o))
        (and (= (CAT$object (category ?o)) (class ?o))
             (= (CAT$morphism (category ?o)) (extent ?o))
             (= (CAT$source (category ?o)) (REL$first ?o))
             (= (CAT$target (category ?o)) (REL$second ?o))))
```

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

```
(8) (KIF$collection partial-order)
    (KIF$subcollection partial-order preorder)
    (forall (?o (preorder ?o))
        (<=> (partial-order ?o)
             (antisymmetric ?o)))
```

- The *identity* endorelation on any class A is a partial order.

```
(9) (KIF$function identity)
    (= (KIF$source identity) class)
    (= (KIF$target identity) partial-order)
    (KIF$restriction identity REL.ENDO$identity)
```

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

```
(10) (KIF$function power)
     (= (KIF$source power) class)
     (= (KIF$target power) partial-order)
     (forall (?c (class ?c))
         (and (= (class (power ?c)) (SET$power ?c))
              (forall (?c1 (SET$subclass ?c1 ?c)
                       ?c2 (SET$subclass ?c2 ?c))
                  (<=> ((power ?c) ?c1 ?c2)
                       (SET$subclass ?c1 ?c2)))))
```

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

```
(11) (KIF$collection total-order)
     (KIF$subcollection total-order partial-order)
     (forall (?o (partial-order ?o))
         (<=> (total-order ?o)
              (forall (?x1 ((class ?o) ?x1)
                       ?x2 ((class ?o) ?x2))
                  (or (?o ?x1 ?x2) (?o ?x2 ?x1)))))
```

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

```
(12) (KIF$function up)
     (= (KIF$source up) partial-order)
     (= (KIF$target up) SET.FTN$function)
     (forall (?o (partial-order ?o))
         (and (SET.FTN$source (up ?o)) (SET$power (class ?o)))
              (SET.FTN$target (up ?o)) (SET$power (class ?o)))
              (forall (?q (SET$subclass ?q (class ?o)) ?x ((class ?o) ?x))
                  (<=> (((up ?o) ?q) ?x)
                       (exists (?y (?q ?y))
                           (?o ?y ?x))))))

(13) (KIF$function down)
     (= (KIF$source down) partial-order)
     (= (KIF$target down) SET.FTN$function)
     (forall (?o (partial-order ?o))
         (and (SET.FTN$source (down ?o)) (SET$power (class ?o)))
              (SET.FTN$target (down ?o)) (SET$power (class ?o)))
              (forall (?q (SET$subclass ?q (class ?o)) ?x ((class ?o) ?x))
                  (<=> (((down ?o) ?q) ?x)
                       (exists (?y (?q ?y))
                           (?o ?x ?y))))))
```

- Clearly, the elements down below a subclass $Q \subseteq P$ in ***P*** are the same elements up above Q in $\boldsymbol{P}^{\mathrm{op}}$.

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

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

```
(14) (KIF$function down-class)
```

```
        (= (KIF$source down-class) partial-order)
        (= (KIF$target down-class) SET$class)
        (forall (?o (partial-order ?o))
            (and (SET$subclass (down-class ?o) (SET$power (class ?o)))
                 (forall (?c (SET$subclass ?c (class ?o)))
                     (<=> ((down-class ?o) ?c)
                          (forall (?x (?c ?x) ?y (?o ?y))
                              (=> (?o ?y ?x) (?c ?y)))))))

   (15) (KIF$function up-class)
        (= (KIF$source up-class) partial-order)
        (= (KIF$target up-class) SET$class)
        (forall (?o (partial-order ?o))
            (and (SET$subclass (up-class ?o) (SET$power (class ?o)))
                 (forall (?c (SET$subclass ?c (class ?o)))
                     (<=> ((up-class ?o) ?c)
                          (forall (?x (?c ?x) ?y (?o ?y))
                              (=> (?o ?x ?y) (?c ?y)))))))
```

- Clearly, a subclass $Q \subseteq P$ is a down class in $\boldsymbol{P}$ iff Q is an up class in $\boldsymbol{P}^{\text{op}}$.

```
        (forall (?o (partial-order ?o))
            (= (down-class ?o) (up-class (opposite ?o))))
```

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

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

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

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

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

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

```
   (16) (KIF$function up-embedding)
        (= (KIF$source up-embedding) partial-order)
        (= (KIF$target up-embedding) SET.FTN$function)
        (forall (?o (partial-order ?o))
            (and (SET.FTN$source (up-embedding ?o)) (class ?o))
                 (SET.FTN$target (up-embedding ?o)) (SET$power (class ?o)))
                 (= (up-embedding ?o)
                    (SET.FTN$composition (SET.FTN$singleton (class ?o)) (up ?o)))))

   (17) (KIF$function down-embedding)
        (= (KIF$source down-embedding) partial-order)
        (= (KIF$target down-embedding) SET.FTN$function)
        (forall (?o (partial-order ?o))
            (and (SET.FTN$source (down-embedding ?o)) (class ?o))
                 (SET.FTN$target (down-embedding ?o)) (SET$power (class ?o)))
```

```
                (= (down-embedding ?o)
                   (SET.FTN$composition (SET.FTN$singleton (class ?o)) (down ?o)))))
```

- Clearly, the elements down below an element $q \in P$ in $\boldsymbol{P}$ are the same elements up above q in $\boldsymbol{P}^{\mathrm{op}}$.

```
     (forall (?o (partial-order ?o)
              ?q ((class ?o) ?q))
         (= ((down-embedding ?o) ?q)
            ((up-embedding (opposite ?o)) ?q)))
```

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

```
(18) (KIF$function greatest)
     (= (KIF$source greatest) partial-order)
     (= (KIF$target greatest) SET.FTN$function)
     (forall (?o (partial-order ?o))
         (and (= (SET.FTN$source (greatest ?o)) (SET$power (class ?o)))
              (= (SET.FTN$target (greatest ?o)) (SET$power (class ?o)))
              (forall (?q (SET$subclass ?q (class ?o))
                       ?a ((class ?o) ?a))
                  (<=> (((greatest ?o) ?q) ?a)
                       (and (?q ?a)
                            (forall (?x (?q ?x))
                                (?o ?x ?a)))))))

(19) (KIF$function least)
     (= (KIF$source least) partial-order)
     (= (KIF$target least) SET.FTN$function)
     (forall (?o (partial-order ?o))
         (and (= (KIF$source (least ?o) (SET$power (class ?o)))
              (= (KIF$target (least ?o) (SET$power (class ?o)))
              (forall (?q (SET$subclass ?q (class ?o))
                       ?a ((class ?o) ?a))
                  (<=> (((least ?o) ?q) ?a)
                       (and (?q ?a)
                            (forall (?x (?q ?x))
                                (?o ?a ?x)))))))
```

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

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

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

- Clearly, the least element of any subclass $Q \subseteq P$ in $\boldsymbol{P}$ is the greatest element of Q in $\boldsymbol{P}^{\mathrm{op}}$.

```
     (forall (?o (partial-order ?o)
              ?q (SET$subclass ?q (class ?o)))
         (= ((least ?o) ?q)
            ((greatest (opposite ?o)) ?q)))
```

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

```
(20) (KIF$function upper-bound)
     (= (KIF$source upper-bound) ORD$partial-order)
     (= (KIF$target upper-bound) SET.FTN$function)
     (forall (?o (ORD$partial-order ?o))
         (and (SET.FTN$source (upper-bound ?o)) (SET$power (class ?o)))
```

```
                (SET.FTN$target (upper-bound ?o)) (SET$power (class ?o)))
                (forall (?q (SET$subclass ?q (ORD$class ?o)) ?y ((ORD$class ?o) ?y))
                    (<=> (((upper-bound ?o) ?q) ?y)
                         (forall (?x (?q ?x))
                             (?o ?x ?y))))))
```

```
(21) (KIF$function lower-bound)
     (= (KIF$source lower-bound) ORD$partial-order)
     (= (KIF$target lower-bound) SET.FTN$function)
     (forall (?o (ORD$partial-order ?o))
         (and (SET.FTN$source (lower-bound ?o)) (SET$power (class ?o)))
              (SET.FTN$target (lower-bound ?o)) (SET$power (class ?o)))
              (forall (?q (SET$subclass ?q (ORD$class ?o)) ?x ((ORD$class ?o) ?x))
                  (<=> (((lower-bound ?o) ?q) ?y)
                       (forall (?x (?q ?x))
                           (?o ?y ?x))))))
```

- Clearly, the lower bounds of a subclass $Q \subseteq P$ in $\boldsymbol{P}$ are the upper bonds of Q in $\boldsymbol{P}^{\mathrm{op}}$.

```
     (forall (?o (partial-order ?o)
              ?q (SET$subclass ?q (class ?o)))
         (= ((lower-bound ?o) ?q)
            ((upper-bound (opposite ?o)) ?q)))
```

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

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

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

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

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

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

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

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

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

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

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

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

```
            (forall (?x (SET$subclass ?x (ORD$class ?o)))
                (SET$subclass ?x ((upper-lower-closure ?o) ?x)))))

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

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

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

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

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

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

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

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

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

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

Examples

- The ordinal *three* is the total order generated by the endorelation *three*.

```
     (total-order three)
     (= three (REL.ENDO$closure REL.ENDO$three))
```

Monotonic Functions

`ORD.FTN`

Preorders and partial orders are most simply related through monotonic functions.

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

  if $x_1 \leq_{\boldsymbol{P}} x_2$ then $f(x_1) \leq_{\boldsymbol{Q}} f(x_2)$ for all $x_1, x_2 \in P$.

```
(1) (KIF$collection monotonic-function)

(2) (KIF$function source)
    (= (KIF$source source) monotonic-function)
    (= (KIF$target source) ORD$preorder)

(3) (KIF$function target)
    (= (KIF$source target) monotonic-function)
    (= (KIF$target target) ORD$preorder)

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

(5) (forall (?f (monotonic-function ?f)
             ?x1 ((ORD$class (source ?f)) ?x1)
             ?x2((ORD$class (source ?f)) ?x2)
        (=> ((source ?f) ?x1 ?x2)
            ((target ?f) (?f ?x1) (?f ?x2))))
```

- Any monotonic function $f : \boldsymbol{P} \rightarrow \boldsymbol{Q}$ defines a *extent* function $\mathit{ext}(f) : \mathit{ext}(\boldsymbol{P}) \rightarrow \mathit{ext}(\boldsymbol{Q})$ between the extent classes of the source and target preorders.

```
(6) (KIF$function extent)
    (= (KIF$source extent) monotonic-function)
    (= (KIF$target extent) SET.FTN$function)
    (forall (?f (monotonic-function ?f))
        (and (= (SET.FTN$source (extent ?f)) (ORD$extent (source ?f)))
             (= (SET.FTN$target (extent ?f)) (ORD$extent (target ?f)))
             (forall (?x ?y (ORD$extent (source ?f)) [?x ?y]))
                 (= ((extent ?f) [?x ?y])
                    [((function ?f) ?x) ((function ?f) ?y)]))))
```

- Any monotonic function is a *functor*.

```
(7) (KIF$function functor)
    (= (KIF$source functor) monotonic-function)
    (= (KIF$target functor) FUNC$functor)
    (forall (?f (monotonic-function ?f))
        (and (= (FUNC$source (functor ?f)) (ORD$category (source ?f)))
             (= (FUNC$target (functor ?f)) (ORD$category (target ?f)))
             (= (FUNC$object (functor ?f)) (function ?f))
             (= (FUNC$morphism (functor ?f)) (extent ?f))))
```

- This conversion from a monotonic function to a functor is quasi-functorial – it preserves composition and identities. We state these theorems in an external namespace.

```
     (forall (?f1 (ORD.FTN$monotonic-function ?f1)
              ?f2 (ORD.FTN$monotonic-function ?f2)
                  (ORD.FTN$composable ?f1 ?f2))
         (= (ORD.FTN$functor (ORD.FTN$composition [?f1 ?f2]))
            (FUNC$composition [(ORD.FTN$functor ?f1) (ORD.FTN$functor ?f2)])))

     (forall (?o (ORD$preorder ?o))
         (= (ORD.FTN$functor (ORD.FTN$identity ?o))
            (FUNC$identity (ORD$category ?o))))
```

- For any two preorders $\boldsymbol{P}$ and $\boldsymbol{Q}$ that are ordered by the suborder relationship $\boldsymbol{P} \subseteq \boldsymbol{Q}$, there is an *inclusion* monotonic function $\boldsymbol{P} \rightarrow \boldsymbol{Q}$.

```
(8) (KIF$function inclusion)
    (= (KIF$source inclusion) (KIF$extent ORD$suborder))
    (= (KIF$target inclusion) monotonic-function)
    (forall (?p (ORD$preorder ?p)
             ?q (ORD$preorder ?q) (ORD$suborder ?p ?q))
        (and (= (source (inclusion [?p ?q])) ?p)
             (= (target (inclusion [?p ?q])) ?q)
             (= (function (inclusion [?p ?q]))
                (SET.FTN$inclusion [(ORD$class ?p) (ORD$class ?q)]))))
```

- One monotonic function $f : \boldsymbol{P} \rightarrow \boldsymbol{Q}$ is a *subfunction* of another monotonic function $g : \boldsymbol{P} \rightarrow \boldsymbol{Q}$ when they share the same source and target preorders and for any element $x \in P$ of the source preorder the images are ordered $f(x) \leq_{\boldsymbol{Q}} g(x)$.

```
(9) (KIF$relation subfunction)
    (= (KIF$collection1 subfunction) monotonic-function)
    (= (KIF$collection2 subfunction) monotonic-function)
    (forall (?f (monotonic-function ?f)
             ?g (monotonic-function ?g))
        (<=> (subfunction ?f ?g)
             (and (= (source ?f) (source ?g))
                  (= (target ?f) (target ?g))
                  (forall (?p ((ORD$class (source ?f)) ?p))
                      ((target ?f) (?f ?p) (?g ?p))))))
```

- Any pair of monotonic functions $\langle f : \boldsymbol{P} \rightarrow \boldsymbol{Q}, g : \boldsymbol{P} \rightarrow \boldsymbol{Q}\rangle$ in a subfunction relationship defines a *sub*function-*extent* function $\mathsf{ext}(f, g) : P \rightarrow \mathsf{ext}(\boldsymbol{Q})$ from the underlying class of the source preorder to the extent class of the target preorder.

```
(10) (KIF$function subextent)
     (= (KIF$source subextent) (KIF$extent subfunction))
     (= (KIF$target subextent) SET.FTN$function)
     (forall (?f (ORD.FTN$monotonic-function ?f)
              ?g (ORD.FTN$monotonic-function ?g)
                 (ORD.FTN$subfunction ?f ?g))
         (and (= (SET.FTN$source (subextent [?f ?g])) (ORD$class (source ?f)))
              (= (SET.FTN$target (subextent [?f ?g])) (ORD$extent (target ?f)))
              (forall (?x ((ORD$class (source ?f)) ?x))
                  (= ((subextent [?f ?g]) ?x)
                     [((function ?f) ?x) ((function ?g) ?x)])))))
```

- Any pair of monotonic functions in the subfunction relationship defines a *natural transformation*.

```
(11) (KIF$function natural-transformation)
     (= (KIF$source natural-transformation) (KIF$extent subfunction))
     (= (KIF$target natural-transformation) NAT$natural-transformation)
     (forall (?f (monotonic-function ?f)
              ?g (monotonic-function ?g) (subfunction ?f ?g))
         (and (= (NAT$source-category (natural-transformation [?f ?g]))
                 (ORD$category (source ?f)))
              (= (NAT$target-category (natural-transformation [?f ?g]))
                 (ORD$category (target ?f)))
              (= (NAT$source-functor (natural-transformation [?f ?g]))
                 (functor ?f))
              (= (NAT$target-functor (natural-transformation [?f ?g]))
                 (functor ?g))
              (= (NAT$component (natural-transformation [?f ?g]))
                 (subextent [?f ?g]))))
```

- One monotonic function $f_1 : \boldsymbol{P}_1 \rightarrow \boldsymbol{Q}_1$ is a *restriction* of another monotonic function $f_2 : \boldsymbol{P}_2 \rightarrow \boldsymbol{Q}_2$ when the source (target) preorder of f_1 is a suborder of the source (target) preorder of f_2 and the underlying function of f_1 is a restriction of the underlying function of f_2.

```
(12) (KIF$relation restriction)
     (= (KIF$collection1 restriction) monotonic-function)
     (= (KIF$collection2 restriction) monotonic-function)
```

```
      (forall (?f1 (monotonic-function ?f1)
               ?f2 (monotonic-function ?f2))
          (<=> (restriction ?f1 ?f2)
               (and (ORD$suborder (source ?f1) (source ?f2))
                    (ORD$suborder (target ?f1) (target ?f2))
                    (SET.FTN$restriction (function ?f1) (function ?f2)))))
```

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

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

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

(15) (KIF$function composition)
     (= (KIF$source composition) (KIF$pullback composable-opspan))
     (= (KIF$target composition) monotonic-function)
     (forall (?f1 (monotonic-function ?f1)
              ?f2 (monotonic-function ?f2) (composable ?f1 ?f2))
         (and (= (source (composition [?f1 ?f2])) (source ?f1))
              (= (target (composition [?f1 ?f2])) (target ?f2))
              (= (function (composition [?f1 ?f2]))
                 (SET.FTN$composition [(function ?f1) (function ?f2)]))))
```

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

```
(16) (KIF$function identity)
     (= (KIF$source identity) ORD$preorder)
     (= (KIF$target identity) monotonic-function)
     (forall (?o (ORD$preorder ?o))
         (and (= (source (identity ?o)) ?o)
              (= (target (identity ?o)) ?o)
              (= (function (identity ?o)) (SET.FTN$identity (ORD$class ?o)))))
```

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

```
(17) (KIF$function opposite)
     (= (KIF$source opposite) monotonic-function)
     (= (KIF$target opposite) monotonic-function)
     (forall (?f (monotonic-function ?f))
         (and (= (source (opposite ?f)) (ORD$opposite (source ?f)))
              (= (target (opposite ?f)) (ORD$opposite (target ?f)))
              (= (function (opposite ?f)) (function ?f))))
```

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

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

  and the *right* bimodule $f^{@} : \boldsymbol{B} \rightarrow \boldsymbol{A}$ as follows

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

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

```
(18) (KIF$function left)
     (= (KIF$source left) (KIF$pullback [target (KIF$identity ORD$preorder)]))
     (= (KIF$target left) BIMOD$bimodule)
     (forall (?f (monotonic-function ?f) ?o (ORD$preorder ?o) (= (target ?f) ?o))
         (and (= (BIMOD$source (left [?f ?o])) ?o)
              (= (BIMOD$target (left [?f ?o])) (source ?f))
              (= (BIMOD$relation (left [?f ?o]))
                 (SET.FTN$left [(function ?f) ?o]))))

(19) (KIF$function right)
     (= (KIF$source right) (KIF$pullback [target (KIF$identity ORD$preorder)]))
     (= (KIF$target right) BIMOD$bimodule)
     (forall (?f (monotonic-function ?f) ?o (ORD$preorder ?o) (= (target ?f) ?o))
         (and (= (BIMOD$source (right [?f ?o])) (source ?f))
              (= (BIMOD$target (right [?f ?o])) ?o)
              (= (BIMOD$relation (right [?f ?o]))
                 (SET.FTN$right [(function ?f) ?o]))))
```

Order Relations

ORD.REL

Preorders are related through order-closed relations known as bimodules. Order bimodules extend binary relations to the order realm.

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

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

  if pRq_1 and $q_1 \leq_{\boldsymbol{Q}} q_2$ then pRq_2 for all elements $p \in \boldsymbol{P}$ and $q_1, q_2 \in \boldsymbol{Q}$.

```
(1) (KIF$collection bimodule)

(2) (KIF$function source)
    (= (KIF$source source) bimodule)
    (= (KIF$target source) ORD$preorder)

(3) (KIF$function target)
    (= (KIF$source target) bimodule)
    (= (KIF$target target) ORD$preorder)

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

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

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

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

```
(7) (KIF$function opposite)
```

```
        (= (KIF$source opposite) bimodule)
        (= (KIF$target opposite) bimodule)
        (forall (?r (bimodule ?r))
            (and (= (source (opposite ?r)) (ORD$opposite (target ?r)))
                 (= (target (opposite ?r)) (ORD$opposite (source ?r)))
                 (= (relation (opposite ?r)) (REL$opposite (relation ?r)))))
```

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

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

(9) (KIF$relation composable)
    (= (KIF$collection1 composable) bimodule)
    (= (KIF$collection2 composable) bimodule)
    (= (KIF$extent composable) (KIF$pullback composable-opspan))

(10) (KIF$function composition)
     (= (KIF$source composition) (KIF$pullback composable-opspan))
     (= (KIF$target composition) bimodule)
     (forall (?r1 (bimodule ?r1) ?r2 (bimodule ?r2) (composable ?r1 ?r2))
         (and (= (source (composition [?r1 ?r2])) (source ?r1))
              (= (target (composition [?r1 ?r2])) (target ?r2))
              (= (relation (composition [?r1 ?r2]))
                 (REL$composition [(relation ?r1) (relation ?r2)]))))
```

- For any preorder $\boldsymbol{P}$ there is an identity order bimodule $id(\boldsymbol{P}) : \boldsymbol{P} \rightarrow \boldsymbol{P}$.

```
(11) (KIF$function identity)
     (= (KIF$source identity) ORD$preorder)
     (= (KIF$target identity) bimodule)
     (forall (?o (ORD$preorder ?o))
         (and (= (source (identity ?o)) ?o)
              (= (target (identity ?o)) ?o)
              (= (relation (identity ?o))
                 (REL$identity (ORD$class ?o)))))
```

Adjoint Pairs

ORD.ADJ

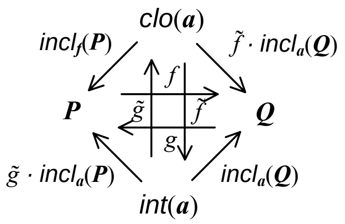


Figure 1: Adjoint Pair

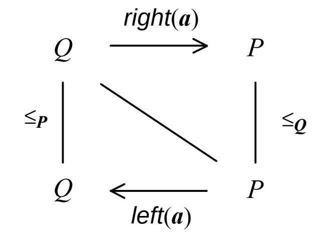


Figure 2: Adjoint Pair as an Infomorphism

- An *adjoint pair* $\boldsymbol{a} : \boldsymbol{P} \rightleftarrows \boldsymbol{Q}$ from preorder $\boldsymbol{P}$ to preorder $\boldsymbol{Q}$ (Figure 1) is a pair $\boldsymbol{a} = \langle f, g \rangle = \langle left(\boldsymbol{a}), right(\boldsymbol{a}) \rangle$ of oppositely directed monotonic functions, $f = left(\boldsymbol{a}) : \boldsymbol{P} \rightarrow \boldsymbol{Q}$ and $g = right(\boldsymbol{a}) : \boldsymbol{Q} \rightarrow \boldsymbol{P}$, which satisfy the *fundamental property*:

  $$f(p) \leq_{\boldsymbol{Q}} q \text{ iff } p \leq_{\boldsymbol{P}} g(q)$$

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

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

  – An adjoint pair $\boldsymbol{a} : \boldsymbol{P} \rightleftarrows \boldsymbol{Q}$ is an infomorphism $\boldsymbol{a} : \boldsymbol{Q} \rightleftarrows \boldsymbol{P}$ from classification $\boldsymbol{Q} = \langle Q, Q, \leq_{\boldsymbol{Q}} \rangle$ to classification $\boldsymbol{P} = \langle P, P, \leq_{\boldsymbol{P}} \rangle$ (Figure 2), where the preorders are regarded as classifications.

```
(1) (KIF$collection adjoint-pair)

(2) (KIF$function source)
    (= (KIF$source source) adjoint-pair)
    (= (KIF$target source) ORD$preorder)
```

```
(3) (KIF$function target)
    (= (KIF$source target) adjoint-pair)
    (= (KIF$target target) ORD$preorder)

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

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

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

○ The fundamental property can be expressed in terms of order on monotonic functions:
- $id_P \leq f \cdot g$
- $g \cdot f \leq id_Q$.

```
(forall (?a (adjoint-pair ?a))
    (and (ORD.FTN$subfunction
             (ORD.FTN$identity (source ?a))
             (ORD.FTN$composition [(left ?a) (right ?a)]))
         (ORD.FTN$subfunction
             (ORD.FTN$composition [(right ?a) (left ?a)])
             (ORD.FTN$identity (target ?a)))))
```

○ The *source closure* operator for an adjoint pair $\boldsymbol{a} : \boldsymbol{P} \rightleftarrows \boldsymbol{Q}$ is the monotonic function $s : \boldsymbol{P} \to \boldsymbol{P}$, defined as the composition $s = f \cdot g$. Dually, the *target closure* operator is the monotonic function $\boldsymbol{t} : \boldsymbol{Q} \to \boldsymbol{Q}$, defined as the composition $t = g \cdot f$.

```
(7) (KIF$function source-closure)
    (= (KIF$source source-closure) adjoint-pair)
    (= (KIF$target source-closure) ORD.FTN$monotonic-function)
    (forall (?a (adjoint-pair ?a))
        (and (= (ORD.FTN$source (source-closure ?a)) (source ?a))
             (= (ORD.FTN$target (source-closure ?a)) (source ?a))
             (= (source-closure ?a)
                (ORD.FTN$composition [(left ?a) (right ?a)]))))

(8) (KIF$function target-closure)
    (= (KIF$source target-closure) adjoint-pair)
    (= (KIF$target target-closure) ORD.FTN$monotonic-function)
    (forall (?a (adjoint-pair ?a))
        (and (= (ORD.FTN$source (target-closure ?a)) (target ?a))
             (= (ORD.FTN$target (target-closure ?a)) (target ?a))
             (= (target-closure ?a)
                (ORD.FTN$composition [(right ?a) (left ?a)]))))
```

○ The source closure monotonic function satisfies the following properties: for all elements $p \in P$,

$p \leq_{\boldsymbol{P}} s(p)$
$s(s(p)) \leq_{\boldsymbol{P}} s(p)$.

Hence, $\boldsymbol{S} = \langle \boldsymbol{P}, \boldsymbol{s} \rangle$ is a *closure system* for the adjoint pair $\boldsymbol{a} : \boldsymbol{P} \rightleftarrows \boldsymbol{Q}$.

The target closure monotonic function satisfies the following properties: for all elements $q \in Q$,

$t(q) \leq_{\boldsymbol{Q}} q$
$s(q) \leq_{\boldsymbol{Q}} s(s(q))$.

Hence, $\boldsymbol{T} = \langle \boldsymbol{P}, \boldsymbol{t} \rangle$ is a *co-closure system* for the adjoint pair $\boldsymbol{a} : \boldsymbol{P} \rightleftarrows \boldsymbol{Q}$.

```
(9) (KIF$function closure-operator)
    (= (KIF$source closure-operator) adjoint-pair)
    (= (KIF$target closure-operator) ORD.CLO$closure-operator)
    (forall (?a (adjoint-pair ?a))
        (and (= (ORD.CLO$preorder (closure-operator ?a)) (source ?a))
             (= (ORD.CLO$closure (closure-operator ?a)) (source-closure ?a))))

(10) (KIF$function co-closure-operator)
     (= (KIF$source co-closure-operator) adjoint-pair)
     (= (KIF$target co-closure-operator) ORD.CLO$co-closure-operator)
     (forall (?a (adjoint-pair ?a))
         (and (= (ORD.CLO$preorder (co-closure-operator ?a)) (target ?a))
              (= (ORD.CLO$co-closure (co-closure-operator ?a)) (target-closure ?a))))
```

○ An element $p \in P$ is *source-closed* with respect to an adjoint pair $\boldsymbol{a} : \boldsymbol{P} \rightleftarrows \boldsymbol{Q}$ when $g(f(p)) = s(p) = p$. The *preorder* of *source-closed* elements *src-clo*($\boldsymbol{a}$) is a suborder of the source preorder $\boldsymbol{P}$.

An element $q \in Q$ is *target-closed* with respect to an adjoint pair $\boldsymbol{a} : \boldsymbol{P} \rightleftarrows \boldsymbol{Q}$ when $f(g(q)) = t(q) = q$. The *preorder* of *target-closed* elements *tgt-clo*($\boldsymbol{a}$) is a suborder of the target preorder $\boldsymbol{Q}$.

```
(11) (KIF$function source-closed-preorder)
     (= (KIF$source source-closed-preorder) adjoint-pair)
     (= (KIF$target source-closed-preorder) ORD$preorder)
     (forall (?a (adjoint-pair ?a))
         (= (source-closed-preorder ?a)
            (ORD.CLO$closed-preorder (closure-operator ?a))))

(12) (KIF$function target-closed-preorder)
     (= (KIF$source target-closed-preorder) adjoint-pair)
     (= (KIF$target target-closed-preorder) ORD$preorder)
     (forall (?a (adjoint-pair ?a))
         (= (target-closed-preorder ?a)
            (ORD.CLO$co-closed-preorder (co-closure-operator ?a))))
```

o The *left closed* monotonic function is the restriction of the left monotonic function to the source and target closed preorders. Dually, the *right closed* monotonic function is the restriction of the right monotonic function to the source and target closed preorders.

```
(13) (KIF$function left-closed)
     (= (KIF$source left-closed) adjoint-pair)
     (= (KIF$target left-closed) ORD.FTN$monotonic-function)
     (forall (?a (adjoint-pair ?a))
         (and (= (ORD.FTN$source (left-closed ?a)) (source-closed-preorder ?a))
              (= (ORD.FTN$target (left-closed ?a)) (target-closed-preorder ?a))
              (ORD.FTN$restriction (left-closed ?a) (left ?a))))

(14) (KIF$function right-closed)
     (= (KIF$source right-closed) adjoint-pair)
     (= (KIF$target right-closed) ORD.FTN$monotonic-function)
     (forall (?a (adjoint-pair ?a))
         (and (= (ORD.FTN$source (right-closed ?a)) (target-closed-preorder ?a))
              (= (ORD.FTN$target (right-closed ?a)) (source- closed-preorder ?a))
              (ORD.FTN$restriction (right-closed ?a) (right ?a))))
```

o That these two functions are well-defined is easy to check. It is also easy to check that these two monotonic functions are inverse to each other. So they form an adjoint pair themselves.

```
     (forall (?a (adjoint-pair ?a))
        (and (= (ORD.FTN$composition [(left-closed ?a) (right-closed ?a)])
                (ORD.FTN$identity (source ?a)))
             (= (ORD.FTN$composition [(right-closed ?a) (left-closed ?a)])
                (ORD.FTN$identity (target ?a)))))
```

o Any adjoint pair of monotonic functions is an *infomorphism*.

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

○ Any adjoint pair of monotonic functions defines an *adjunction*.

```
(8) (KIF$function adjunction)
    (= (KIF$source adjunction) adjoint-pair)
    (= (KIF$target adjunction) ADJ$adjunction)
    (forall (?a (adjoint-pair ?a))
        (and (= (ADJ$underlying-category (adjunction ?a))
                (ORD$category (source ?a)))
             (= (ADJ$free-category (adjunction ?a))
                (ORD$category (target ?a)))
             (= (ADJ$left-adjoint (adjunction ?a))
                (ORD.FTN$functor (left ?a)))
             (= (ADJ$right-adjoint (adjunction ?a))
                (ORD.FTN$functor (right ?a)))
             (= (ADJ$unit (adjunction ?a))
                (ORD.FTN$natural-transformation
                    [(ORD.FTN$identity (source ?a))
                     (ORD.FTN$composition [(left ?a) (right ?a)])]))
             (= (ADJ$counit (adjunction ?a))
                (ORD.FTN$natural-transformation
                    [(ORD.FTN$composition [(right ?a) (left ?a)])
                     (ORD.FTN$identity (source ?a))])))))
```

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

```
(9) (KIF$function bond)
    (= (KIF$source bond) adjoint-pair)
    (= (KIF$target bond) CLS.BND$bond)
    (forall (?a (adjoint-pair ?a))
        (and (= (CLS.BND$source (bond ?a)) (ORD$classification (target ?a)))
             (= (CLS.BND$target (bond ?a)) (ORD$classification (source ?a)))
             (= (CLS.BND$classification (bond ?a))
                (CLS.INFO$bond (infomorphism ?a)))))
```

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

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

o Two adjoint pairs $\boldsymbol{f}_1 : \boldsymbol{O} \rightleftarrows \boldsymbol{P}$ and $\boldsymbol{f}_2 : \boldsymbol{P} \rightleftarrows \boldsymbol{Q}$ are *composable* when the target order of $\boldsymbol{f}_1$ is the same as the source order of $\boldsymbol{f}_2$. There is a KIF function *composition*, which takes two composable adjoint pairs and returns their composition.

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

(12) (KIF$relation composable)
     (= (KIF$collection1 composable) adjoint-pair)
     (= (KIF$collection2 composable) adjoint-pair)
     (= (KIF$extent composable) (KIF$pullback composable-opspan))

(13) (KIF$function composition)
     (= (KIF$source composition) (KIF$pullback composable-opspan))
     (= (KIF$target composition) adjoint-pair)
```

```
          (forall (?f1 (adjoint-pair ?f1)
                   ?f2 (adjoint-pair ?f2) (composable ?f1 ?f2))
              (and (= (source (composition [?f1 ?f2])) (source ?f1))
                   (= (target (composition [?f1 ?f2])) (target ?f2))
                   (= (left (composition [?f1 ?f2]))
                      (ORD.FTN$composition [(left ?f1) (left ?f2)]))
                   (= (right (composition [?f1 ?f2]))
                      (ORD.FTN$composition [(right ?f2) (right ?f1)]))))
```

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

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

- Any adjoint pair $\boldsymbol{f} : \boldsymbol{P} \rightleftarrows \boldsymbol{Q}$ has an associated *closure operator* $\langle \boldsymbol{P}, \mathit{left}(\boldsymbol{f}) \cdot \mathit{right}(\boldsymbol{f})\rangle$, whose preorder is the source of f and whose closure is the composite of the left and right adjoints.

```
(15) (KIF$function closure-operator)
     (= (KIF$source identity) ORD$preorder)
     (= (KIF$target identity) adjoint-pair)
     (forall (?f (adjoint-pair ?f))
         (and (= (preorder (closure-operator ?f)) (source ?f))
              (= (closure (closure-operator ?f))
                 (ORD.FTN$composition [(left ?f) (right ?f)]))))
```

Closure Operators

ORD.CLO

- An (abstract) *closure structure* (*closure operator*) is a pair $\boldsymbol{C} = \langle \boldsymbol{A}, \boldsymbol{c}\rangle$ consisting of a preorder $\boldsymbol{A} = \langle A, \leq_A\rangle$ and a monotonic function $\boldsymbol{c} : \boldsymbol{A} \rightarrow \boldsymbol{A}$, which satisfy the following fundamental properties:

  $id_A \leq c$
  $c \cdot c \ \leq id_A$.

  Using elements this can be expressed as follows: for all elements $a \in A$,

  $a \leq_A c(a)$
  $c(c(a)) \leq_A c(a)$.

  Note: the second inequality is an equality for partial orders, but not necessarily for preorders.

```
(1) (KIF$collection closure-structure)
    (KIF$collection closure-operator)
    (= closure-operator closure-structure)

(2) (KIF$function preorder)
    (= (KIF$source preorder) closure-structure)
    (= (KIF$target preorder) ORD$preorder)

(3) (KIF$function closure)
    (= (KIF$source closure) closure-structure)
    (= (KIF$target closure) ORD.FTN$monotonic-function)
    (forall (?cs (closure-structure ?cs))
        (and (= (ORD.FTN$source (closure ?cs)) (ORD$class (preorder ?cs)))
             (= (ORD.FTN$target (closure ?cs)) (ORD$class (preorder ?cs)))
             (forall (?a ((ORD$class (preorder ?cs)) ?a))
                 (and ((preorder ?cs) ?a ((ORD.FTN$function (closure ?cs)) ?a))
                      ((preorder ?cs)
                          ((ORD.FTN$function (closure ?cs))
                              ((ORD.FTN$function (closure ?cs)) ?a))
                          ((ORD.FTN$function (closure ?cs)) ?a))))))
```

- An element $a \in A$ is *closed* with respect to a closure operator $\boldsymbol{c} : \boldsymbol{A} \rightarrow \boldsymbol{A}$ when $c(a) = a$.

```
(4) (KIF$function closed)
```

```
        (= (KIF$source closed) closure-structure)
        (= (KIF$target closed) SET$class)
        (forall (?cs (closure-structure ?cs))
            (and (SET$subclass (closed ?cs) (ORD$class (preorder ?cs)))
                 (forall (?a ((ORD$class (preorder ?cs)) ?a))
                     (<=> ((closed ?cs) ?a)
                          (= ((ORD.FTN$function (closure ?cs)) ?a) ?a)))))
```

- The *preorder* of *closed* elements $\mathsf{clo}_c(\boldsymbol{A})$ is a suborder of the preorder $\boldsymbol{A}$.

```
    (5) (KIF$function closed-preorder)
        (= (KIF$source closed-preorder) closure-structure)
        (= (KIF$target closed-preorder) ORD$preorder)
        (forall (?cs (closure-structure ?cs))
            (and (ORD$suborder (closed-preorder ?cs) (preorder ?cs))
                 (= (ORD$class (closed-preorder ?cs)) (closed ?cs))))
```

- The first condition on a closure operator corresponds to an *extent* function $\mathsf{ext}(\boldsymbol{c}) : A \rightarrow \mathsf{ext}(\boldsymbol{A})$.

```
    (6) (KIF$function extent)
        (= (KIF$source extent) closure-structure)
        (= (KIF$target extent) SET.FTN$function)
        (forall (?cs (closure-structure ?cs))
            (and (SET.FTN$source (extent ?cs)) (ORD$class (preorder ?cs)))
                 (SET.FTN$target (extent ?cs)) (ORD$extent (preorder ?cs)))
                 (forall (?a ((ORD$class (preorder ?cs)) ?a))
                     (= ((extent ?cs) ?a)
                        [?a ((ORD.FTN$function (closure ?cs)) ?a)]))))
```

- Any closure operator defines a *monad*.

```
    (7) (KIF$function monad)
        (= (KIF$source monad) closure-structure)
        (= (KIF$target monad) MND$monad)
        (forall (?cs (closure-structure ?cs))
            (and (= (MND$category (monad ?cs)) (ORD$category (preorder ?cs)))
                 (= (MND$functor (monad ?cs)) (ORD.FTN$functor (closure ?cs)))
                 (= (MND$unit (monad ?cs))
                    (ORD.FTN$natural-transformation
                        [(ORD.FTN$identity (preorder ?cs))
                         (closure ?cs)]))
                 (= (MND$multiplication (monad ?cs))
                    (ORD.FTN$natural-transformation
                        [(ORD.FTN$composition [(closure ?cs) (closure ?cs)])
                         (ORD.FTN$identity (preorder ?cs))]))))
```

- It can be proven that the monad of the closure operator generated by an adjoint pair is the same as the monad generated by the adjunction of the adjoint pair. This is stated in an external namespace.

```
        (forall (?a (ORD.ADJ$adjoint-pair ?a))
            (= (ORD.CLO$monad (ORD.ADJ$closure-operator ?a))
               (ADJ$monad (ORD.ADJ$adjunction ?a))))
```

- Any closure operator $\boldsymbol{c} : \boldsymbol{A} \rightarrow \boldsymbol{A}$ has an associated adjoint pair $\langle \boldsymbol{c}, \mathit{incl} \rangle : \boldsymbol{A} \rightleftarrows \mathsf{clo}_c(\boldsymbol{A})$, whose left adjoint is the closure operator restricted to the closed elements (hence surjective), and whose right adjoint is inclusion.

```
    (8) (KIF$function adjoint-pair)
        (= (KIF$source adjoint-pair) closure-structure)
        (= (KIF$target adjoint-pair) ORD.ADJ$adjoint-pair)
        (forall (?cs (closure-structure ?cs))
            (and (= (ORD.ADJ$source (adjoint-pair ?cs)) (preorder ?cs))
                 (= (ORD.ADJ$target (adjoint-pair ?cs)) (closed-preorder ?cs))
                 (= (ORD.ADJ$left (adjoint-pair ?cs)) (closure ?cs))
                 (= (ORD.ADJ$right (adjoint-pair ?cs))
                    (ORD.FTN$inclusion [(closed-preorder ?cs) (preorder ?cs)]))))
```

- It can be directly proven that the closure operator generated by the adjoint pair of a closure operator is the original closure operator.

```
         (forall (?cs (closure-structure ?cs))
             (= (ORD.ADJ$closure-operator (adjoint-pair ?cs)) ?cs))
```

- An (abstract) *coclosure structure* (*coclosure operator*) (*interior operator*) $\boldsymbol{D} = \langle \boldsymbol{A}, \boldsymbol{d} \rangle$ is a closure structure $\boldsymbol{D} = \langle \boldsymbol{A}^{\mathrm{op}}, \boldsymbol{d}^{\mathrm{op}} \rangle$ on the opposite preorder; that is, A *coclosure operator* is a pair $\boldsymbol{D} = \langle \boldsymbol{A}, \boldsymbol{d} \rangle$ consisting of a preorder $\boldsymbol{A} = \langle A, \leq_{\boldsymbol{A}} \rangle$ and a monotonic function $\boldsymbol{d} : \boldsymbol{A} \rightarrow \boldsymbol{A}$, which satisfy the following fundamental properties:

  $d \leq \mathsf{id}_{\boldsymbol{A}}$
  $\mathsf{id}_{\boldsymbol{A}} \leq d \cdot d.$

  Using elements this can be expressed as follows: for all elements $a \in A$,

  $a \leq_{\boldsymbol{A}} d(a)$
  $d(d(a)) \leq_{\boldsymbol{A}} d(a).$

  Note: the second inequality is an equality for partial orders, but not necessarily for preorders.

The Namespace of Large Graphs

Table 1 lists the terms defined and axiomatized in the namespaces for graphs and graph morphisms.

Table 1: Terms introduced in the Graph Namespace

	Collections	Function	Other

`GPH`	`graph`
	`small`
	`diagram`
		`object = node morphism = edge`
		`source target opposite`
		`shape class function diagram-fiber`
			`empty two three`
			`parallel-pair`
			`opspan span`
			`subgraph`
			`empty-diagram`

		`category = free = path`
		`free-embedding = path-embedding`
		`free-extension = path-extension`
		`path-opspan path-triple evaluation`	

		`multiplication-opspan multiplication unit`	`multipliable-opspan` `multipliable`

`GPH`
`.MOR`
	`graph-morphism`	`source target object morphism`
		`exponent inclusion substitution restriction`
			`multipliable-opspan`
			`multipliable`

		`functor = free`	

		`multiplication-cone`
		`multiplication unit`	

	`2-cell`	`composition identity`
		`isomorphism inverse tau`
			`composable-opspan`
			`composable`
			`isomorphic`

	`invariant`	`graph = base`
		`object-endorelation morphism-endorelation`
		`quotient canon mediator`
			`respects`

		`first-cone opspan12-3second-cone`
		`alpha associativity`
		`left right`	

Table 2: Elements of the 2-dimensional category GRAPH

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

$$\begin{tikzcd}
{} \arrow[r, "G_0"] \arrow[d, "f_0"'] \arrow[dr, phantom, "H_0"] & {} \arrow[r, "G_1"] \arrow[d, "f_1"'] \arrow[dr, phantom, "H_1"] & {} \arrow[d, "f_2"] \\
{} \arrow[r, "G'_0"] \arrow[d, "f'_0"'] \arrow[dr, phantom, "H'_0"] & {} \arrow[r, "G'_1"] \arrow[d, "f'_1"'] \arrow[dr, phantom, "H'_1"] & {} \arrow[d, "f'_2"] \\
{} \arrow[r, "G''_0"'] & {} \arrow[r, "G''_1"'] & {}
\end{tikzcd}$$

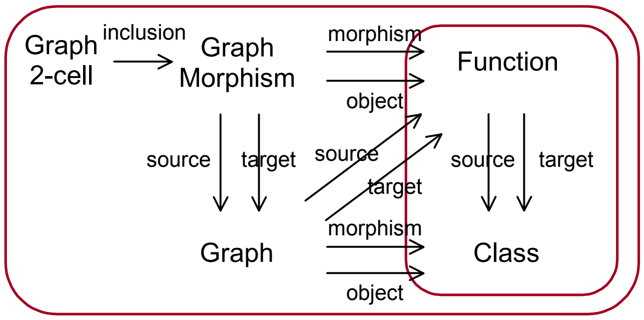


Diagram 1: Core Collections and Functions

Graphs

GPH

$$\begin{tikzcd} mor(\boldsymbol{G}) \arrow[r, shift left, "src(\boldsymbol{G})"] \arrow[r, shift right, "tgt(\boldsymbol{G})"'] & obj(\boldsymbol{G}) \end{tikzcd}$$

Figure 1: Graph

○ A (*large*) *graph* $\boldsymbol{G}$ (Figure 1) consists of
- a class $obj(\boldsymbol{G})$ of *objects* (*nodes*),
- a class $mor(\boldsymbol{G})$ of *morphisms* (arrows or *edges*),
- a *source* (domain) function $src(\boldsymbol{G}) : mor(\boldsymbol{G}) \rightarrow obj(\boldsymbol{G})$ and
- a *target* (codomain) function $tgt(\boldsymbol{G}) : mor(\boldsymbol{G}) \rightarrow obj(\boldsymbol{G})$.

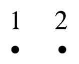
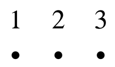
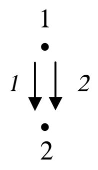
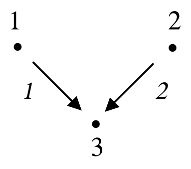
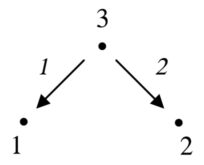
A morphism $m \in mor(\boldsymbol{G})$, with source object $src(\boldsymbol{G})(m) = o_0 \in obj(\boldsymbol{G})$ and target object $tgt(\boldsymbol{G})(m) = o_1 \in obj(\boldsymbol{G})$, is usually represented graphically with the notation $m : o_0 \rightarrow o_1$.

The following is the KIF representation for the elements of a graph. Graphs are uniquely determined by their (object, morphism, source, target) quadruple.

```
(1) (KIF$collection graph)

(2) (KIF$function object)
    (KIF$function node)
    (= object node)
    (= (KIF$source object) graph)
    (= (KIF$target object) SET$class)

(3) (KIF$function morphism)
    (KIF$function edge)
    (= morphism edge)
    (= (KIF$source morphism) graph)
    (= (KIF$target morphism) SET$class)

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

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

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

○ Here are some small examples of graphs that are used as the shapes of finite limit diagrams: for a terminal class (this uses the *empty* shape diagram), for the binary (co)product of *two* classes, for the (co)equalizers of a *parallel pair* of class functions, for the pullback of an *opspan* of classes and class functions, and for the pushout of a *span* of classes and class functions.

Table 1: Shapes for Diagrams

					
empty (terminal) (intitial)	*two* (binary product) (binary coproduct)	*three* (ternary product) (ternary coproduct)	*parallel pair* (equalizer) (coequalizer)	*opspan* (pullback)	*span* (pushout)

```
(6) (graph empty)
    (= (node empty) SET$empty)
    (= (edge empty) SET$empty)
    (= (source empty) (SET.FTN$counique SET$empty))
    (= (target empty) (SET.FTN$counique SET$empty))

(7) (graph two)
    (= (node two) SET$two)
    (= (edge two) SET$empty)
    (= (source two) (SET.FTN$counique SET$two))
    (= (target two) (SET.FTN$counique SET$two))

(8) (graph three)
    (= (node three) SET$three)
    (= (edge three) SET$empty)
    (= (source three) (SET.FTN$counique SET$three))
    (= (target three) (SET.FTN$counique SET$three))

(9) (graph parallel-pair)
    (= (node parallel-pair) SET$two)
    (= (edge parallel-pair) SET$two)
    (= (source parallel-pair) ((SET.FTN$constant [SET$two SET$two]) 1))
    (= (target parallel-pair) ((SET.FTN$constant [SET$two SET$two]) 2))

(10) (graph opspan)
     (= (node opspan) SET$three)
     (= (edge opspan) SET$two)
     (= (source opspan) (SET.FTN$inclusion [SET$two SET$three]))
     (= (target opspan) ((SET.FTN$constant [SET$two SET$three]) 3))

(11) (graph span)
     (= (node span) SET$three)
     (= (edge span) SET$two)
     (= (source opspan) ((SET.FTN$constant [SET$two SET$three]) 3))
     (= (target opspan) (SET.FTN$inclusion [SET$two SET$three]))
```

○ To each graph $\boldsymbol{G}$, there is an *opposite graph* $\boldsymbol{G}^{\mathrm{op}}$. The opposite graph is also called the *dual graph*. The objects of $\boldsymbol{G}^{\mathrm{op}}$ are the objects of $\boldsymbol{G}$, and the morphisms of $\boldsymbol{G}^{\mathrm{op}}$ are the morphisms of $\boldsymbol{G}$. However, the source and target functions are reversed: $src(\boldsymbol{G}^{\mathrm{op}}) = tgt(\boldsymbol{G})$ and $tgt(\boldsymbol{G}^{\mathrm{op}}) = src(\boldsymbol{G})$.

```
(12) (KIF$function opposite)
     (= (KIF$source opposite) graph)
     (= (KIF$target opposite) graph)
     (forall (?g (graph ?g))
         (and (= (object (opposite ?g)) (object ?g))
              (= (morphism (opposite ?g)) (morphism ?g))
```

```
                (= (source (opposite ?g)) (target ?g))
                (= (target (opposite ?g)) (source ?g))))
```

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

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

○ One graph G_1 is a subgraph of another graph G_2 when the object/morphism class of the first is a subclass of the second, and the source/target function of the first is a restriction of the second.

```
(13) (KIF$relation subgraph)
     (= (KIF$collection1 subgraph) graph)
     (= (KIF$collection2 subgraph) graph)
     (forall (?g1 (graph ?g1) ?g2 (graph ?g2))
         (<=> (subgraph ?g1 ?g2)
              (and (SET$subclass (object ?g1) (object ?g2))
                   (SET$subclass (morphism ?g1) (morphism ?g2))
                   (SET.FTN$restriction (source ?g1) (source ?g2))
                   (SET.FTN$restriction (target ?g1) (target ?g2)))))
```

○ The *empty* graph is a subgraph of the other three shape graphs above, and *two* is a subgraph of *opspan*.

```
    (subgraph empty two)
    (subgraph empty parallel-pair)
    (subgraph empty opspan)
    (subgraph two opspan)
```

○ A graph G is small when both the object and morphism classes are (small) sets. This is a concrete concept, since it uses concepts in the lower metalevel.

```
(14) (KIF$collection small)
     (KIF$subcollection small graph)
     (forall (?g (graph ?g))
         (<=> (small ?g)
              (and (set$set (object ?g))
                   (set$set (morphism ?g)))))
```

o A *diagram* is essentially a quasi-graph morphism from a *shape* graph into the implicit quasi-graph of collections and functions. The object (node) function of this quasi-graph morphism is here called *class*, and the morphism (edge) function of this quasi-graph morphism is here called *function*. Diagrams are determined by their *shape*, *class* and *function*. Note that the class of a diagram is a KIF tuple collection.

```
(15) (KIF$collection diagram)

(16) (KIF$function shape)
     (= (KIF$source shape) diagram)
     (= (KIF$target shape) graph)

(17) (KIF$function class)
     (= (KIF$source class) diagram)
     (= (KIF$target class) KIF$function)
     (forall (?d (diagram ?d))
         (and (= (KIF$source (class ?d)) (node (shape ?d)))
              (= (KIF$target (class ?d)) SET$class)))

(18) (KIF$function function)
     (= (KIF$source function) diagram)
     (= (KIF$target function) KIF$function)
     (forall (?d (diagram ?d))
         (and (= (KIF$source (function ?d)) (edge (shape ?d)))
              (= (KIF$target (function ?d)) SET.FTN$function)))
              (forall (?e ((edge (shape ?d)) ?e))
                  (and (= (SET.FTN$source ((function ?d) ?e))
                          ((class ?d) ((source (shape ?d)) ?e)))
                       (= (SET.FTN$target ((function ?d) ?e))
                          ((class ?d) ((target (shape ?d)) ?e)))))))

     (forall (?d1 (diagram ?d1) ?d2 (diagram ?d2))
         (=> (and (= (class1 ?d1) (class1 ?d2))
                  (= (class2 ?d1) (class2 ?d2)))
             (= ?d1 ?d2)))
```

- Consider the empty shape graph. Since the shape is empty, there is only one diagram consisting of the empty class/function functions.

```
(19) (GPH$diagram empty-diagram)
     (= (GPH$shape empty-diagram) GPH$empty)
```

- The *diagram-fiber* *diagram*(*G*) of any graph *G* is the collection of all diagrams of shape *G*.

```
(20) (KIF$function diagram-fiber)
     (= (KIF$source diagram-fiber) GPH$graph))
     (= (KIF$target diagram-fiber) KIF$collection)
     (= diagram-fiber (KIF$fiber shape))
```

Free Category

- Over any graph ***G*** there is
  - a *free* category *free*(***G***) whose underlying graph shares the object class with the original graph,
  - a *free embedding* graph 2-cell η_{G}: ***G*** ⇒ |*free*(***G***)| that is universal for each graph ***G***, and
  - a *free extension* isomorphism $\varphi_{G,C}$: GRAPH[***G***, |***C***|] → CAT[*free*(***G***), ***C***] that is natural in graph ***G*** and category ***C***. We use a definite description to define the free extension function.

```
(21) (KIF$function category)
     (KIF$function free)
     (= free category)
     (= (KIF$source free) graph)
     (= (KIF$target free) CAT$category)
     (forall (?g (graph ?g))
         (= (CAT$object (free ?g)) (object ?g)))

(22) (KIF$function free-embedding)
     (= (KIF$source free-embedding) graph)
     (= (KIF$target free-embedding) GPH.MOR$2-cell)
     (forall (?g (graph ?g))
         (and (= (GPH.MOR$source (free-embedding ?g)) ?g)
              (= (GPH.MOR$target (free-embedding ?g)) (CAT$graph (free ?g)))))

(23) (KIF$function free-extension)
     (= (KIF$source free-extension) (KIF$binary-product [graph CAT$category]))
     (= (KIF$target free-extension) KIF$function)
     (forall (?g (graph ?g) ?c (CAT$category ?c))
         (= (free-extension [?g ?c])
            (the (?f (KIF$function ?f))
                (and (= (KIF$source ?f)
                        (GPH.MOR$exponent [?g (CAT$graph ?c)]))
                     (= (KIF$target ?f)
                        (FUNC$exponent [(free ?g) ?c]))
                     (forall (?h (GPH.MOR$graph-morphism ?h))
                              (= (GPH.MOR$source ?h) ?g)
                              (= (GPH.MOR$target ?h) (CAT$graph ?c)))
                         (= ?h (GPH.MOR$composition
                                   [(free ?g)
                                    (FUNC$graph-morphism (?f ?h))])))))))
```

- Paths in a graph ***G*** are constructively built from morphisms (edges) in ***G***: a path of ***G*** is either an object of ***G***, with source and target being itself; a morphism of ***G***, with source and target given by the graph ***G***, or a composable pair consisting of a morphism of ***G*** and a morphism of *path*(***G***) where the graph target of the first is the path source of the second. Over any graph ***G*** there is
  - a *path* category *path*(***G***) whose underlying graph shares object class with the original graph,
  - a *path embedding* graph 2-cell η_{G}: ***G*** ⇒ |*path*(***G***)|, and
  - a *path extension* isomorphism $\varphi_{G,C}$: GRAPH[***G***, |***C***|] → CAT[*free*(***G***), ***C***] that is natural in graph ***G*** and category ***C***. We define the path extension function recursively.

$$\begin{tikzcd} \text{CATEGORY} \arrow[d, shift left, "\substack{\textit{underlying}\\ \textit{= graph}}"] \\ \text{GRAPH} \arrow[u, shift left, "\substack{\textit{free}\\ \textit{= path}}"] \end{tikzcd}$$

Figure 2: Quasi -adjunction

```
(24) (KIF$function path-opspan)
     (= (KIF$source path-opspan) graph)
     (= (KIF$target path-opspan) SET.LIM.PBK$opspan)
     (forall (?g (graph ?g))
         (and (= (SET.LIM.PBK$class1 (path-opspan ?g)) (morphism ?g))
              (= (SET.LIM.PBK$class2 (path-opspan ?g)) (CAT$morphism (path ?g)))
              (= (SET.LIM.PBK$opvertex (path-opspan ?g)) (object ?g))
              (= (SET.LIM.PBK$opfirst (path-opspan ?g)) (target ?g))
              (= (SET.LIM.PBK$opsecond (path-opspan ?g)) (CAT$source (path ?g)))))

(25) (KIF$function path)
     (= (KIF$source path) graph)
     (= (KIF$target path) CAT$category)
     (forall (?g (graph ?g))
         (and (= (CAT$object (path ?g)) (object ?g))
              (forall (?p)
                  (<=> ((CAT$morphism (path ?g)) ?p)
                       (or ((object ?g) ?p)
                           ((morphism ?g) ?p)
                           ((SET.LIM.PBK$pullback (path-opspan ?g)) ?p))))))

(26) (forall (?g (graph ?g)
              ?p (CAT$morphism (path ?g)) ?p))
         (and (=> ((object ?g) ?p)
                  (and (= ((CAT$source (path ?g)) ?p) ?p)
                       (= ((CAT$target (path ?g)) ?p) ?p)))
              (=> ((morphism ?g) ?p)
                  (and (= ((CAT$source (path ?g)) ?p) ((source ?g) ?p))
                       (= ((CAT$target (path ?g)) ?p) ((target ?g) ?p))))
              (=> ((SET.LIM.PBK$pullback (path-opspan ?g)) ?p)
                  (and (= ((CAT$source (path ?g)) ?p)
                          ((source ?g)
                            ((SET.LIM.PBK$projection1 (path-opspan ?g)) ?p)))
                       (= ((CAT$target (path ?g)) ?p)
                          ((CAT$target (path ?g))
                            ((SET.LIM.PBK$projection2 (path-opspan ?g)) ?p)))))))

(27) (forall (?g (graph ?g)
              ?p (CAT$morphism (path ?g)) ?p)
              ?q (CAT$morphism (path ?g)) ?q)
              ((CAT$composable (path ?g)) ?p ?q))
       (and (=> ((object ?g) ?p)
                (= ((CAT$composition (path ?g)) [?p ?q]) ?q))
            (=> ((morphism ?g) ?p)
                (= ((CAT$composition (path ?g)) [?p ?q]) [?p ?q])
            (=> ((SET.LIM.PBK$pullback (path-opspan ?g)) ?p)
                (= ((CAT$composition (path ?g)) [?p ?q])
                   [((SET.LIM.PBK$projection1 (path-opspan ?g)) ?p)
                    ((CAT$composition (path ?g))
                        [((SET.LIM.PBK$projection2 (path-opspan ?g)) ?p) ?q])]))))

(28) (forall (?g (graph ?g) ?o ((object ?g) ?o))
         (= ((CAT$identity (path ?g)) ?o) ?o))

(29) (KIF$function path-embedding)
     (= (KIF$source path-embedding) graph)
     (= (KIF$target path-embedding) GPH.MOR$2-cell)
     (forall (?g (graph ?g))
         (and (= (GPH.MOR$source (path-embedding ?g)) ?g)
              (= (GPH.MOR$target (path-embedding ?g)) (CAT$graph (path ?g)))))

(30) (KIF$function path-extension)
     (= (KIF$source path-extension) (KIF$binary-product [graph CAT$category]))
     (= (KIF$target path-extension) KIF$function)
     (forall (?g (graph ?g) ?c (CAT$category ?c))
         (and (= (KIF$source (path-extension [?g ?c]))
                 (GPH.MOR$exponent [?g (CAT$graph ?c)]))
              (= (KIF$target (path-extension [?g ?c]))
                 (FUNC$exponent [(path ?g) ?c]))
              (forall (?h (GPH.MOR$graph-morphism ?h))
                       (= (GPH.MOR$source ?h) ?g)
                       (= (GPH.MOR$target ?h) (CAT$graph ?c)))
                  (and (= (FUNC$object ((path-extension [?g ?c]) ?h))
                          (GPH.MOR$object ?h))
                       (forall (?p (CAT$morphism (path ?g)) ?p))
                          (and (=> ((object ?g) ?p)
                                 (= ((FUNC$morphism ((path-extension [?g ?c]) ?h)) ?p)
                                    ((GPH.MOR$object ?h) ?p))
                               (=> ((morphism ?g) ?p)
                                 (= ((FUNC$morphism ((path-extension [?g ?c]) ?h)) ?p)
                                    ((GPH.MOR$morphism ?h) ?p))
                               (=> ((SET.LIM.PBK$pullback (path-opspan ?g)) ?p)
                                 (= ((FUNC$morphism ((path-extension [?g ?c]) ?h)) ?p)
                                    [((GPH.MOR$morphism ?h)
                                         ((SET.LIM.PBK$projection1
                                             (path-opspan ?g)) ?p))
                                     ((FUNC$morphism ((path-extension [?g ?c]) ?h))
                                         (((SET.LIM.PBK$projection2
                                             (path-opspan ?g)) ?p))])))))))
```

- We can alternately define the path class, the morphism class of the path category, as a disjoint union.

```
(31) (KIF$function path-triple)
     (= (KIF$source path-triple) graph)
     (= (KIF$target path-triple) SET.COL.COPRD3$triple)
     (forall (?g (graph ?g))
         (and (= (SET.COL.COPRD3$class1 (path-triple ?g)) (object ?g))
              (= (SET.COL.COPRD3$class2 (path-triple ?g)) (morphism ?g))
              (= (SET.COL.COPRD3$class3 (path-triple ?g))
                 (SET.LIM.PBK$pullback (path-opspan ?g)))))

(32) (forall (?g (graph ?g))
         (= (CAT$morphism (path ?g))
            (SET.COL.COPRD3$ternary-coproduct (path-triple ?g))))
```

- For any graph ***G***, we assert that the free category over ***G*** is the path category over ***G***, the free embedding graph 2-cell is the path embedding 2-cell, and the free extension isomorphism is the path extension isomorphism. These assertions give the free notions a specific concrete representation.

```
(33) (forall (?g (graph ?g))
         (and (= (free ?g) (path ?g))
              (= (free-embedding ?g) (path-embedding ?g))
              (forall (?c (CAT$category ?c))
                  (= (free-extension [?g ?c]) (path-extension [?g ?c])))))
```

- For any category ***C***, there is an *evaluation* functor $\varepsilon_{\boldsymbol{C}} : path(|\boldsymbol{C}|) \rightarrow \boldsymbol{C}$ whose source is the free category over the underlying graph of ***C*** and whose target is ***C***. This is define in terms of the composition and identities in ***C***. The functor $\varepsilon_{\boldsymbol{C}}$ is the ***C***-th component of the counit of the quasi-adjunction (Figure 2) between the free category functor and the underlying graph functor.

```
(34) (KIF$function evaluation)
     (= (KIF$source evaluation) CAT$category)
     (= (KIF$target evaluation) FUNC$functor)
     (forall (?c (CAT$category ?c))
         (and (= (FUNC$source (evaluation ?c)) (path (CAT$graph ?c)))
              (= (FUNC$target (evaluation ?c)) ?c)
              (= (FUNC$object (evaluation ?c)) (SET.FTN$identity (CAT$object ?c)))
              (forall (?p (CAT$morphism (path ?g)) ?p))
                  (and (=> ((object ?g) ?p)
                           (= ((FUNC$morphism (evaluation ?c)) ?p)
                              ((CAT$identity ?c) ?p)))
                       (=> ((morphism ?g) ?p)
                           (= ((FUNC$morphism (evaluation ?c)) ?p) ?p))
                       (=> ((SET.LIM.PBK$pullback (path-opspan ?g)) ?p)
                           (= ((FUNC$morphism (evaluation ?c)) ?p)
                              ((CAT$composition ?c)
                                  [((SET.LIM.PBK$projection1 (path-opspan ?g)) ?p)
                                   ((FUNC$morphism (evaluation ?c))
                                       ((SET.LIM.PBK$projection2
                                           (path-opspan ?g)) ?p))])))))))
```

Multiplication

○ Two graphs $\boldsymbol{G}_0$ and $\boldsymbol{G}_1$ are *horizontally composable* (*multipliable*) when they share a class of objects $obj(\boldsymbol{G}_0) = obj = obj(\boldsymbol{G}_1)$. For two composable graphs there is an associated *multiplication* graph $\boldsymbol{G}_0 \otimes \boldsymbol{G}_1$ (Figure 3), whose class of morphisms is the class of *composable pairs* of morphisms defined above. This is the pullback in foundations along the target function $tgt(\boldsymbol{G}_0) : mor(\boldsymbol{G}_0) \rightarrow obj$ and the source function $src(\boldsymbol{G}_1) : mor(\boldsymbol{G}_1) \rightarrow obj$.

$$\begin{tikzcd}
 & & mor(\boldsymbol{G}_0)\times_{obj}mor(\boldsymbol{G}_1) \arrow[dl, "1st"'] \arrow[dr, "2nd"] & & \\
 & mor(\boldsymbol{G}_0) \arrow[dl, "src(\boldsymbol{G}_0)"'] \arrow[dr, "tgt(\boldsymbol{G}_0)"] & & mor(\boldsymbol{G}_1) \arrow[dl, "src(\boldsymbol{G}_1)"'] \arrow[dr, "tgt(\boldsymbol{G}_1)"] & \\
obj & & obj & & obj
\end{tikzcd}$$

Figure 3: Multiplication Graph

$mor(\boldsymbol{G}_0) \times_{obj} mor(\boldsymbol{G}_1) =$
$\{(m_0, m_1) \mid m_0 \in mor(\boldsymbol{G}_0) \text{ and } m_1 \in mor(\boldsymbol{G}_1) \text{ and } tgt(\boldsymbol{G}_0)(m_0) = src(\boldsymbol{G}_1)(m_1)\}$

Each horizontally-composable pair of graphs has an auxiliary foundational opspan represented as the term '`(multiplication-opspan ?g0 ?g1)`'. This opspan allows us to refer to the appropriate foundational pullback. The multiplication graph $G_0 \otimes G_1$ is represented as the term '`(multiplication ?g0 ?g1)`'.

```
(35) (KIF$opspan multipliable-opspan)
     (= composable-opspan [object object])

(36) (KIF$relation multipliable)
     (= (KIF$collection1 multipliable) graph)
     (= (KIF$collection2 multipliable) graph)
     (= (KIF$extent multipliable) (KIF$pullback multipliable-opspan))

(37) (KIF$function multiplication-opspan)
     (= (KIF$source multiplication-opspan) (KIF$pullback multipliable-opspan))
     (= (KIF$target multiplication-opspan) SET.LIM.PBK$opspan)
     (forall (?g0 (graph ?g0) ?g1 (graph ?g1) (multipliable ?g0 ?g1))
         (and (= (SET.LIM.PBK$opvertex (multiplication-opspan [?g0 ?g1]))
                 (object ?g0))
              (= (SET.LIM.PBK$opfirst (multiplication-opspan [?g0 ?g1]))
                 (target ?g0))
              (= (SET.LIM.PBK$opsecond (multiplication-opspan [?g0 ?g1]))
                 (source ?g1))))

(38) (KIF$function multiplication)
     (= (KIF$source multiplication) (KIF$pullback multipliable-opspan))
     (= (KIF$target multiplication) graph)
     (forall (?g0 (graph ?g0) ?g1 (graph ?g1) (multipliable ?g0 ?g1))
         (and (= (object (multiplication [?g0 ?g1]))
                 (object ?g0))
              (= (morphism (multiplication [?g0 ?g1]))
                 (SET.LIM.PBK$pullback (multiplication-opspan [?g0 ?g1])))
              (= (source (multiplication [?g0 ?g1]))
                 (SET.FTN$composition
                     [(SET.LIM.PBK$projection1 (multiplication-opspan [?g0 ?g1]))
                      (source ?g0)]))
              (= (target (multiplication [?g0 ?g1]))
                 (SET.FTN$composition
                     [(SET.LIM.PBK$projection2 (multiplication-opspan [?g0 ?g1]))
                      (target ?g1)]))))
```

Unit

○ For any class (of objects) C there is a *unit* graph 1_C (Figure 4) whose classes of objects and morphisms are C, and whose source and target functions is the SET identity function on C. The unit graph has the following formalization.

```
(39) (KIF$function unit)
     (= (KIF$source unit) SET$class)
     (= (KIF$target unit) graph)
```

$$\begin{tikzcd}
 & C \arrow[dl, "id_C"'] \arrow[dr, "id_C"] & \\
C & & C
\end{tikzcd}$$

$$\begin{tikzcd}
C \arrow[r, shift left, "id_C"] \arrow[r, shift right, "id_C"'] & C
\end{tikzcd}$$

Figure 4: Unit Graph

```
        (forall (?c (SET$class ?c))
            (and (= (object (unit ?c)) ?c)
                 (= (morphism (unit ?c)) ?c)
                 (= (source (unit ?c)) (SET.FTN$identity ?c))
                 (= (target (unit ?c)) (SET.FTN$identity ?c))))
```

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

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

Graph Morphisms

GPH.MOR

$$\begin{tikzcd} mor(\boldsymbol{G}) \arrow[r, shift left, "src(\boldsymbol{G})"] \arrow[r, shift right, "tgt(\boldsymbol{G})"'] \arrow[d, "mor(\boldsymbol{H})"'] & obj(\boldsymbol{G}) \arrow[d, "obj(\boldsymbol{H})"] \\ mor(\boldsymbol{G}') \arrow[r, shift left, "src(\boldsymbol{G}')"] \arrow[r, shift right, "tgt(\boldsymbol{G}')"'] & obj(\boldsymbol{G}') \end{tikzcd}$$

Figure 5: Graph Morphism

○ A *graph morphism* $\boldsymbol{H} : \boldsymbol{G} \Rightarrow \boldsymbol{G}'$ (Figure 5) from graph $\boldsymbol{G}$ to graph $\boldsymbol{G}'$ consists of two functions, an *object* function and a *morphism* function, that preserve source and target (the diagram in Figure 4 is commutative). The object function $obj(\boldsymbol{H}) : obj(\boldsymbol{G}) \rightarrow obj(\boldsymbol{G}')$ assigns to each object of $\boldsymbol{G}$ an object of $\boldsymbol{G}'$, and the morphism function $mor(\boldsymbol{H}) : mor(\boldsymbol{G}) \rightarrow mor(\boldsymbol{G}')$ assigns to each morphism of $\boldsymbol{G}$ a morphism of $\boldsymbol{G}'$. In the graph morphism that is presented in Figure 4, the diagram is asserted to be commutative. What this means is that the component functions must preserve source and target in the sense that the following constraints must be satisfied.

$$mor(\boldsymbol{H}) \cdot src(\boldsymbol{G}') = src(\boldsymbol{G}) \cdot obj(\boldsymbol{H}) \text{ and } mor(\boldsymbol{H}) \cdot tgt(\boldsymbol{G}') = tgt(\boldsymbol{G}) \cdot obj(\boldsymbol{H})$$

The following is a formalization of a graph morphism. Graph morphisms are uniquely determined by their (source, target, object, morphism) quadruple.

```
(1) (KIF$collection graph-morphism)

(2) (KIF$function source)
    (= (KIF$source source) graph-morphism)
    (= (KIF$target source) GPH$graph)

(3) (KIF$function target)
    (= (KIF$source target) graph-morphism)
    (= (KIF$target target) GPH$graph)

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

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

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

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

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

```
(7) (KIF$function opposite)
    (= (KIF$source opposite) graph-morphism)
    (= (KIF$target opposite) graph-morphism)
    (forall (?h (graph-morphism ?h))
        (and (= (source (opposite ?h)) (GPH$opposite (source ?h)))
             (= (target (opposite ?h)) (GPH$opposite (target ?h)))
             (= (object (opposite ?h)) (object ?h))
             (= (morphism (opposite ?h)) (morphism ?h))))
```

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

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

○ For any pair of graphs $\boldsymbol{G}_1$ and $\boldsymbol{G}_2$ in there is an *exponent* collection of graph morphisms whose source graph is $\boldsymbol{G}_1$ and whose target graph is $\boldsymbol{G}_2$.

```
(8) (KIF$function exponent)
    (= (KIF$source exponent) (KIF$binary-product [GPH$graph GPH$graph]))
    (= (KIF$target exponent) KIF$collection)
    (forall (?g1 (GPH$graph ?g1) ?g2 (GPH$graph ?g2))
        (and (KIF$subcollection (exponent [?g1 ?g2]) graph-morphism)
             (forall (?h (graph-morphism ?h))
                 (<=> ((exponent [?g1 ?g2]) ?h)
                      (and (= (source ?h) ?g1)
                           (= (target ?h) ?g2))))))
```

○ To every pair of graphs $\boldsymbol{G}_1$ and $\boldsymbol{G}_2$ in a subgraph relationship, there is an *inclusion* graph morphism $\boldsymbol{G}_1 \Rightarrow \boldsymbol{G}_2$ from the subgraph to the supergraph.

```
(9) (KIF$function inclusion)
    (= (KIF$source inclusion) (KIF$extent GPH$subgraph))
    (= (KIF$target inclusion) graph-morphism)
    (forall (?g1 (graph ?g1) ?g2 (graph ?g2) (GPH$subgraph ?g1 ?g2))
        (and (= (source (inclusion [?g1 ?g2])) ?g1)
             (= (target (inclusion [?g1 ?g2])) ?g2)
             (= (object (inclusion [?g1 ?g2]))
                (SET.FTN$inclusion [(object ?g1) (object ?g2)]))
             (= (morphism (inclusion [?g1 ?g2]))
                (SET.FTN$inclusion [(morphism ?g1) (morphism ?g2)]))))
```

o Associated with any graph morphism $\boldsymbol{H} : \boldsymbol{G}_1 \Rightarrow \boldsymbol{G}_2$ is a *substitution* function from the diagram fiber of $\boldsymbol{G}_2$ to the diagram fiber of $\boldsymbol{G}_1$.

```
(10) (KIF$function substitution)
     (= (KIF$source substitution) GPH.MOR$graph-morphism))
     (= (KIF$target substitution) KIF$function)
     (forall (?h (graph-morphism ?h))
         (and (KIF$source (substitution ?h)) (diagram-fiber (GPH.MOR$target ?h)))
              (KIF$target (substitution ?h)) (diagram-fiber (GPH.MOR$source ?h)))
              (forall (?d ((diagram-fiber (GPH.MOR$target ?h)) ?d))
                  (and (shape ((substitution ?h) ?d)) (GPH.MOR$source ?h))
                       (forall (?n ((node (GPH.MOR$source ?h)) ?n))
                           (= ((class ((substitution ?h) ?d)) ?n)
                              ((class ?d) ((node ?h) ?n)))
                       (forall (?e ((edge (GPH.MOR$source ?h)) ?e))
                           (= ((function ((substitution ?h) ?d)) ?e)
                              ((function ?d) ((edge ?h) ?e)))))))
```

- As a special case, if the graph G_1 is a subgraph of graph G_2, then there is a *restriction* function from the diagram fiber of G_2 to the diagram fiber of G_1. This is just the composition of the inclusion operation for graph morphisms with the above substitution.

```
(11) (KIF$function restriction)
     (= (KIF$source restriction) (KIF$extent GPH$subgraph))
     (= (KIF$target restriction) KIF$function)
     (forall (?g1 (GPH$graph ?g1) ?g2 (GPH$graph ?g2) (GPH$subgraph ?g1 ?g2))
         (and (KIF$source (restriction [?g1 ?g2])) (diagram-fiber ?g2))
              (KIF$target (restriction [?g1 ?g2])) (diagram-fiber ?g1))
              (= (restriction [?g1 ?g2])
                 (substitution (GPH.MOR$inclusion [?g1 ?g2])))))
```

- Over any graph morphism $H : G \Rightarrow G'$ there is
  - a *free* functor $\mathit{free}(H): \mathit{free}(G) \rightarrow \mathit{free}(G')$ whose underlying graph morphism shares the object function with the original graph morphism.

  This is defined in terms of the free extension with respect to G of the graph morphism composition $H \circ \eta_{G'} : G \Rightarrow |\mathit{free}(G')|$.

```
(12) (KIF$function functor)
     (KIF$function free)
     (= free functor)
     (= (KIF$source free) graph-morphism)
     (= (KIF$target free) FUNC$functor)
     (forall (?h (graph-morphism ?h))
         (and (= (FUNC$source (free ?h)) (GPH$free (GPH.MOR$source ?h)))
              (= (FUNC$target (free ?h)) (GPH$free (GPH.MOR$target ?h)))
              (= (free ?h)
                 ((GPH$free-extension [(source ?h) (GPH$free (target ?h))])
                      (composition [?h (GPH$free-embedding (target ?h))])))))
```

- This conversion from a graph morphism to a functor is quasi-functorial – it preserves composition and identities. We state these theorems in an external namespace.

```
     (forall (?h1 (GPH.MOR$graph-morphism ?h1)
              ?h2 (GPH.MOR$graph-morphism ?h2)
                  (GPH.MOR$composable ?h1 ?h2))
         (= (GPH.MOR$free (GPH.MOR$composition [?h1 ?h2]))
            (FUNC$composition [(GPH.MOR$free ?h1) (GPH.MOR$free ?h2)])))

     (forall (?g (GPH$graph ?g))
         (= (GPH.MOR$free (GPH.MOR$identity ?g))
            (FUNC$identity (GPH$free ?g))))
```

Multiplication

- The *multiplication* operation on graphs can be extended to graph morphisms. For any two graphs morphisms $H_0 : G_0 \Rightarrow G'_0$ and $H_1 : G_1 \Rightarrow G'_1$, which are *horizontally composable* (*multipliable*), in that they share a common object function $\mathit{obj}(H_0) = \mathit{obj} = \mathit{obj}(H_1)$, there is a *multiplication* graph morphism $H_1 \otimes H_2 : G_0 \otimes G_1 \Rightarrow G'_0 \otimes G'_1$, (Figure 6) whose object function is the common object function and whose morphism function is determined by pullback along the morphism functions of H_0 and H_1.

  The formalism for the multiplication of graphs used an auxiliary associated pullback diagram (opspan). In comparison and contrast, the formalism for the multiplication of graph morphisms uses an auxiliary pullback *cone*.

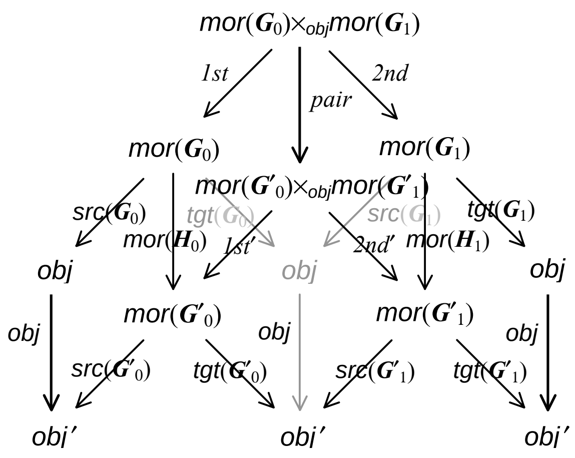


Figure 6: Multiplication Graph Morphism

```
    (13) (KIF$opspan multipliable-opspan)
         (= composable-opspan [object object])

    (14) (KIF$relation multipliable)
         (= (KIF$collection1 multipliable) graph-morphism)
         (= (KIF$collection2 multipliable) graph-morphism)
         (= (KIF$extent multipliable) (KIF$pullback multipliable-opspan))

    (15) (KIF$function multiplication-cone)
         (= (KIF$source multiplication-cone) (KIF$pullback multipliable-opspan))
         (= (KIF$target multiplication-cone) SET.LIM.PBK$cone)
         (forall (?h0 (graph-morphism ?h0)
                  ?h1 (graph-morphism ?h1) (multipliable ?h0 ?h1))
             (and (= (SET.LIM.PBK$cone-diagram (multiplication-cone [?h0 ?h1]))
                     (GPH$multiplication-opspan [(target ?h0) (target ?h1)]))
                  (= (SET.LIM.PBK$vertex (multiplication-cone [?h0 ?h1]))
                     (SET.LIM.PBK$pullback
                         (GPH$multiplication-opspan [(source ?h0) (source ?h1)])))
                  (= (SET.LIM.PBK$first (multiplication-cone [?h0 ?h1]))
                     (SET.FTN$composition
                         [(SET.LIM.PBK$projection1
                              [(GPH$multiplication-opspan [(source ?h0) (source ?h1)]))
                          (morphism ?h0)]))
                  (= (SET.LIM.PBK$second (multiplication-cone [?h0 ?h1]))
                     (SET$composition
                         [(SET.LIM.PBK$projection2
                              (GPH$multiplication-opspan [(source ?h0) (source ?h1)]))
                          (morphism ?h1)]))))

    (16) (KIF$function multiplication)
         (= (KIF$source multiplication) (KIF$pullback multipliable-opspan))
         (= (KIF$target multiplication) graph-morphism)
         (forall (?h0 (graph-morphism ?h0)
                  ?h1 (graph-morphism ?h1) (multipliable ?h0 ?h1))
             (and (= (source (multiplication [?h0 ?h1]))
                     (GPH$multiplication [(source ?h0) (source ?h1)]))
                  (= (target (multiplication [?h0 ?h1]))
                     (GPH$multiplication [(target ?h0) (target ?h1)]))
                  (= (object (multiplication [?h0 ?h1]))
                     (object ?h0))
                  (= (morphism (multiplication ?h0 ?h1))
                     (SET.LIM.PBK$mediator (multiplication-cone [?h0 ?h1])))))
```

Unit

○ For any function (of objects) $f : A \to B$ there is a *unit* graph morphism (Figure 7) $1_f : 1_A \Rightarrow 1_B$, whose source and target graphs are the unit graphs for A and B, and whose object and morphism functions are f.

$$\begin{tikzcd}
A \arrow[r, shift left, "Id_A"] \arrow[r, shift right, "Id_A"'] \arrow[d, "f"'] & A \arrow[d, "f"] \\
B \arrow[r, shift left, "Id_B"] \arrow[r, shift right, "Id_B"'] & B
\end{tikzcd}$$

Figure 7: Unit Graph Morphism

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

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

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

2-Dimensional Category Structure

○ A pair of graph morphisms $\boldsymbol{H}_1$ and $\boldsymbol{H}_2$ is (*vertically*) *composable* when the target graph of $\boldsymbol{H}_1$ is the source graph of $\boldsymbol{H}_2$. For any composable pair of graph morphisms $\boldsymbol{H}_1 : \boldsymbol{G}_0 \Rightarrow \boldsymbol{G}_1$ and $\boldsymbol{H}_2 : \boldsymbol{G}_1 \Rightarrow \boldsymbol{G}_2$, there is a (*vertical*) *composition* graph morphism $\boldsymbol{H}_1 \circ \boldsymbol{H}_2 : \boldsymbol{G}_0 \Rightarrow \boldsymbol{G}_2$. Its object and morphism functions are the compositions of the object and morphism functions of the component graph morphisms. Composition satisfies the typing constraints

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

for all composable pairs of graph morphisms H_1 and H_2.

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

(19) (KIF$relation composable)
     (= (KIF$collection1 composable) graph-morphism)
     (= (KIF$collection2 composable) graph-morphism)
     (= (KIF$extent composable) (KIF$pullback composable-opspan))

(20) (KIF$function composition)
     (= (KIF$source composition) (KIF$pullback composable-opspan))
     (= (KIF$target composition) graph-morphism)
     (forall (?h1 (graph-morphism ?h1) ?h2 (graph-morphism ?h2) (composable ?h1 ?h2))
         (and (= (source (composition [?h1 ?h2])) (source ?h1))
              (= (target (composition [?h1 ?h2])) (target ?h2))
              (= (object (composition [?h1 ?h2]))
                 (SET.FTN$composition [(object ?h1) (object ?h2)]))
              (= (morphism (composition [?h1 ?h2]))
                 (SET.FTN$composition [(morphism ?h1) (morphism ?h2)]))))
```

For any graph G there is an *identity* graph morphism $\mathsf{id}_{\boldsymbol{G}} : \boldsymbol{G} \Rightarrow \boldsymbol{G}$ on that graph. Its object and morphism functions are the identity functions on the object and morphism classes of that graph, respectively. Identity satisfies the following typing constraints

$$src(id_G) = G = tgt(id_G)$$

for all graphs G.

```
(21) (KIF$function identity)
     (= (KIF$source identity) GPH$graph)
     (= (KIF$target identity) graph-morphism)
     (forall (?g (GPH$graph ?g))
         (and (= (source (identity ?g)) ?g)
              (= (target (identity ?g)) ?g)
              (= (object (identity ?g))
                 (SET.FTN$identity (GPH$object ?g)))
              (= (morphism (identity ?g))
                 (SET.FTN$identity (GPH$morphism ?g)))))
```

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

$$(\boldsymbol{H}_1 \circ \boldsymbol{H}_2) \circ \boldsymbol{H}_3 = \boldsymbol{H}_1 \circ (\boldsymbol{H}_2 \circ \boldsymbol{H}_3)$$

for all composable pairs of graph morphisms $(\boldsymbol{H}_1, \boldsymbol{H}_2)$ and $(\boldsymbol{H}_2, \boldsymbol{H}_3)$, and graph morphism identity satisfies the following identity laws

$$\mathit{Id}_{\boldsymbol{G}0} \cdot \boldsymbol{H} = \boldsymbol{H} \text{ and } \boldsymbol{H} = \boldsymbol{H} \cdot \mathit{Id}_{\boldsymbol{G}1}$$

for any graph morphism $\boldsymbol{H} : \boldsymbol{G}_0 \Rightarrow \boldsymbol{G}_1$ with source graph $\boldsymbol{G}_0$ and target graph $\boldsymbol{G}_1$. This has the following expression in an external namespace.

```
(forall (?h1 (GPH.MOR$graph-morphism ?h1)
         ?h2 (GPH.MOR$graph-morphism ?h2)
         ?h3 (GPH.MOR$graph-morphism ?h3))
    (=> (and (GPH.MOR$composable ?h1 ?h2)
             (GPH.MOR$composable ?h2 ?h3))
        (= (GPH.MOR$composition [(GPH.MOR$composition [?h1 ?h2]) ?h3])
           (GPH.MOR$composition [?h1 (GPH.MOR$composition [?h2 ?h3])]))))

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

Graphs as objects and graph morphisms as morphisms form the quasi-category GRAPH. The prefix "quasi" is used, since this is at the level of generic collections in foundations)

○ Two oppositely directed graph morphisms $\boldsymbol{H} : \boldsymbol{G}_0 \rightarrow \boldsymbol{G}_1$ and $\boldsymbol{H}' : \boldsymbol{G}_1 \rightarrow \boldsymbol{G}_0$ are *inverses* of each other when $\boldsymbol{H} \circ \boldsymbol{H}' = id_{\boldsymbol{G}0}$ and $\boldsymbol{H}' \circ \boldsymbol{H} = id_{\boldsymbol{G}1}$. A *graph isomorphism* is a graph morphism that has an inverse. With these laws, we can prove the theorem that an inverse to a graph morphism is unique. The *inverse* function maps an isomorphism to its inverse (another isomorphism). This is a bijection. Two graphs are said to be *isomorphic* when there is a graph isomorphism between them.

```
(22) (KIF$collection isomorphism)
     (KIF$subcollection isomorphism graph-morphism)
     (forall (?h (graph-morphism ?h))
         (<=> (isomorphism ?h)
              (exists (?h1 (graph-morphism ?h1))
                  (and (= (composable ?h ?h1)
                       (= (composable ?h1 ?h)
                       (= (composition [?h ?h1]) (identity (source ?h)))
                       (= (composition [?h1 ?h]) (identity (target ?h)))))))

(23) (KIF$function inverse)
     (= (KIF$source inverse) isomorphism)
     (= (KIF$target inverse) isomorphism)
     (forall (?h (isomorphism ?h))
         (= (inverse ?h)
            (the (?h1 (isomorphism ?h1))
                (and (= (composable ?h ?h1)
                     (= (composable ?h1 ?h)
                     (= (composition [?h ?h1]) (identity (source ?h)))
                     (= (composition [?h1 ?h]) (identity (target ?h)))))))

(24) (KIF$relation isomorphic)
     (= (KIF$collection1 isomorphic) GPH$graph)
     (= (KIF$collection2 isomorphic) GPH$graph)
     (forall ?g1 (GPH$graph ?g1) ?g2 (GPH$graph ?g2))
         (<=> (isomorphic ?g1 ?g2)
              (exists (?h)
                  (and (isomorphism ?h)
                       (= (source ?h) ?g1)
                       (= (target ?h) ?g2)))))
```

○ Given a graph $\boldsymbol{G} = \langle obj(\boldsymbol{G}), mor(\boldsymbol{G}), src(\boldsymbol{G}), tgt(\boldsymbol{G})\rangle$, an *invariant* is a pair $J = \langle \boldsymbol{O}, \boldsymbol{M}\rangle$ consisting of an *object endorelation* $O \subseteq obj(\boldsymbol{G}) \times obj(\boldsymbol{G})$ and a *morphism endorelation* $M \subseteq mor(\boldsymbol{G}) \times mor(\boldsymbol{G})$ that satisfies the *fundamental constraints*:

if $m_0\, M\, m_1$ then $src(\boldsymbol{G})(m_0)\ O\ src(\boldsymbol{G})(m_1)$ and $tgt(\boldsymbol{G})(m_0)\ O\ tgt(\boldsymbol{G})(m_1)$

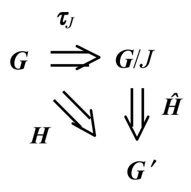
for each $m_0, m_1 \in mor(\boldsymbol{G})$. The graph $\boldsymbol{G}$ is called the *base graph* of J – the graph on which J is based. An invariant is determined by its base, object endorelation and morphism endorelation triple.

```
(25) (KIF$collection invariant)

(26) (KIF$function graph)
     (KIF$function base)
     (= base graph)
     (= (KIF$source graph) invariant)
     (= (KIF$target graph) GPH$graph)

(27) (KIF$function object-endorelation)
     (= (KIF$source object-endorelation) invariant)
     (= (KIF$target object-endorelation) REL.ENDO$endorelation)

(28) (KIF$function morphism-endorelation)
     (= (KIF$source morphism-endorelation) invariant)
```

```
          (= (KIF$target morphism-endorelation) REL.ENDO$endorelation)

     (29) (forall (?j (invariant ?j))
              (and (= (REL.ENDO$class (object-endorelation ?j) (GPH$object (graph ?j)))
                   (= (REL.ENDO$class (morphism-endorelation ?j)) (GPH$morphism (graph ?j)))
                   (forall (?m0 ?m1 ((morphism-endorelation ?j) ?m0 ?m1))
                       (and ((object-endorelation ?j)
                                ((GPH$source (graph ?j)) ?m0)
                                ((GPH$source (graph ?j)) ?m1))
                            ((object-endorelation ?j)
                                ((GPH$source (graph ?j)) ?m0)
                                ((GPH$source (graph ?j)) ?m1))))))

         (forall (?j1 (coinvariant ?j1) ?j2 (coinvariant ?j2))
             (=> (and (= (graph ?j1) (graph ?j2))
                      (= (object-endorelation ?j1) (object-endorelation ?j2))
                      (= (morphism-endorelation ?j1) (morphism-endorelation ?j2)))
                 (= ?j1 ?j2)))
```

○ Often, the endorelations $\boldsymbol{O}$ and $\boldsymbol{M}$ on objects and morphisms are equivalence relations. However, it is convenient not to require this. The endorelation $\boldsymbol{O}$ ($\boldsymbol{M}$) is contained in a smallest equivalence relation $\equiv_{\boldsymbol{O}}$ ($\equiv_{\boldsymbol{M}}$) on objects (types) called the equivalence relation generated by $\boldsymbol{O}$ ($\boldsymbol{M}$). This is the reflexive, symmetric, transitive closure of $\boldsymbol{O}$. For any object $o \in obj(\boldsymbol{G})$ (morphism $m \in mor(\boldsymbol{G})$), write $[o]_{\boldsymbol{O}}$ ($[m]_{\boldsymbol{M}}$) for the $\boldsymbol{O}$-equivalence class of o ($\boldsymbol{M}$-equivalence class of m). Then $\equiv_{\boldsymbol{J}} = \langle \equiv_{\boldsymbol{O}}, \equiv_{\boldsymbol{M}} \rangle$ is also an invariant on $\boldsymbol{G}$. $\boldsymbol{H}_0 : \boldsymbol{G}_0 \Rightarrow \boldsymbol{G}'_0$

The *quotient* $\boldsymbol{G}/J$ of an invariant J on a graph $\boldsymbol{G}$ (Figure 8 is the graph defined as follows:



$$\begin{tikzcd} \boldsymbol{G} \arrow[r, Rightarrow, "\boldsymbol{\tau}_J"] \arrow[dr, Rightarrow, "\boldsymbol{H}"'] & \boldsymbol{G}/J \arrow[d, Rightarrow, "\hat{\boldsymbol{H}}"] \\ & \boldsymbol{G}' \end{tikzcd}$$

Figure 8: Universality of the quotient

– The class of objects of $\boldsymbol{G}/J$ is $obj(\boldsymbol{G})/\boldsymbol{O}$, the quotient class over $obj(\boldsymbol{G})$ of $\boldsymbol{O}$-equivalence classes.
– The class of morphisms of $\boldsymbol{G}/J$ is $mor(\boldsymbol{G})/\boldsymbol{M}$, the quotient class over $mor(\boldsymbol{G})$ of $\boldsymbol{M}$-equivalence classes.
– a *source* function $src(\boldsymbol{G}/J) : mor(\boldsymbol{G}/J) \rightarrow obj(\boldsymbol{G}/J)$ is defined by

$$src(\boldsymbol{G}/J)([m]_M) = [src(\boldsymbol{G})(m)]_O$$

– a *target* (codomain) function $tgt(\boldsymbol{G}) : mor(\boldsymbol{G}) \rightarrow obj(\boldsymbol{G})$ is defined by

$$tgt(\boldsymbol{G}/J)([m]_M) = [tgt(\boldsymbol{G})(m)]_O.$$

where both source and target functions are well-defined by the fundamental condition.

There is a *canonical* quotient infomorphism $\boldsymbol{\tau}_J : \boldsymbol{G} \rightleftarrows \boldsymbol{G}/J$, whose object (node) function is the canonical quotient function $[\text{-}]_{\boldsymbol{O}} : obj(\boldsymbol{G}) \rightarrow obj(\boldsymbol{G})/\boldsymbol{O}$, and whose morphism (edge) function is the canonical quotient function $[\text{-}]_{\boldsymbol{M}} : mor(\boldsymbol{G}) \rightarrow mor(\boldsymbol{G})/\boldsymbol{M}$. The fundamental property for this graph morphism is trivial, given the definition of the quotient source and target functions above.

```
     (30) (KIF$function quotient)
          (= (KIF$source quotient) invariant)
          (= (KIF$target quotient) GPH$graph)
          (forall (?j (invariant ?j))
              (and (= (CLS$instance (coquotient ?j)) (class ?j))
                   (= (GPH$object (quotient ?j))
                      (REL.ENDO$quotient
                          (REL.ENDO$equivalence-closure (object-endorelation ?j))))
                   (= (GPH$morphism (quotient ?j))
                      (REL.ENDO$quotient
                          (REL.ENDO$equivalence-closure (morphism-endorelation ?j))))
                   (= (GPH$source (quotient ?j))
                      (SET.FTN$composition
                          [(GPH$source (graph ?j))
                           (REL.ENDO$canon
                             (REL.ENDO$equivalence-closure (object-endorelation ?j)))]))
                   (= (GPH$target (quotient ?j))
                      (SET.FTN$composition
                          [(GPH$target (graph ?j))
                           (REL.ENDO$canon
                         (REL.ENDO$equivalence-closure (object-endorelation ?j)))])))))
```

```
(31) (KIF$function canon)
     (= (KIF$source canon) invariant)
     (= (KIF$target canon) GPH.MOR$graph-morphism)
     (forall (?j (invariant ?j))
         (and (= (GPH.MOR$source (canon ?j)) (graph ?j))
              (= (GPH.MOR$target (canon ?j)) (quotient ?j))
              (= (GPH.MOR$object (canon ?j))
                 (SET.FTN$inclusion [(class ?j) (CLS$instance (base ?j))]))
              (= (GPH.MOR$object (canon ?j))
                 (REL.ENDO$canon
                     (REL.ENDO$equivalence-closure (object-relation ?j))))
              (= (GPH.MOR$morphism (canon ?j))
                 (REL.ENDO$canon
                     (REL.ENDO$equivalence-closure (morphism-relation ?j))))))
```

- Let $J = \langle \boldsymbol{O}, \boldsymbol{M} \rangle$ be an invariant on a graph $\boldsymbol{G}$. A graph morphism $\boldsymbol{H} : \boldsymbol{G} \Rightarrow \boldsymbol{G}'$ *respects* J when:
  - for any two objects $o_0, o_1 \in obj(\boldsymbol{G})$, if $(o_0, o_1) \in O$ then $obj(\boldsymbol{H})(o_0) = obj(\boldsymbol{H})(o_1)$; and
  - for any two morphisms $m_0, m_1 \in mor(\boldsymbol{G})$, if $(m_0, m_1) \in M$ then $mor(\boldsymbol{H})(m_0) = mor(\boldsymbol{H})(m_1)$.

```
(32) (KIF$relation respects)
     (= (KIF$collection1 respects) GPH.MOR$graph-morphism)
     (= (KIF$collection2 respects) invariant)
     (forall (?h (GPH.MOR$graph-morphism ?h)
              ?j (invariant ?j))
         (<=> (respects ?h ?j)
              (and (= (GPH.MOR$source ?h) (graph ?j))
                   (forall (?o0 ((GPH$object (GPH.MOR$source ?h)) ?o0)
                            ?o1 ((GPH$object (GPH.MOR$source ?h)) ?o1)
                            ((object-endorelation ?j) ?o0 ?o1))
                       (= ((GPH.MOR$object ?h) ?o0)
                          ((GPH.MOR$object ?h) ?o1)))
                   (forall (?m0 ((GPH$morphism (GPH.MOR$source ?h)) ?m0)
                            ?m1 ((GPH$morphism (GPH.MOR$source ?h)) ?m1)
                            ((morphism-endorelation ?j) ?t0 ?t1))
                       (= ((GPH.MOR$morphism ?h) ?m0)
                          ((GPH.MOR$morphism ?h) ?m1))))))
```

Proposition. *For every invariant J on a graph* $\boldsymbol{G}$ *and every graph morphism* $\boldsymbol{H} : \boldsymbol{G} \Rightarrow \boldsymbol{G}'$ *that respects J, there is a unique mediating graph morphism* $\hat{\boldsymbol{H}} : \boldsymbol{G}/J \Rightarrow \boldsymbol{G}'$ *such that* $\tau_J \circ \hat{\boldsymbol{H}} = \boldsymbol{H}$ *(the diagram in Figure 7 commutes).*

Based on this proposition, a definite description is used to define a *mediator* function (Figure 7) that maps a pair $(\boldsymbol{H}, J)$ consisting of an invariant and a respectful graph morphism to their mediator $\hat{\boldsymbol{H}}$.

```
(33) (KIF$function mediator)
     (= (KIF$source mediator) (KIF$extent respects))
     (= (KIF$target mediator) GPH.MOR$graph-morphism)
     (forall (?j (invariant ?j)
              ?h (GPH.MOR$graph-morphism ?h) (respects ?h ?j))
         (= (mediator [?h ?j])
            (the (?ht (GPH.MOR$graph-morphism ?ht))
                (and (= (GPH.MOR$source ?ht) (quotient ?j))
                     (= (GPH.MOR$target ?ht) (GPH.MOR$target ?h))
                     (= (GPH.MOR$composition [(canon ?j) ?ht]) ?h)))))
```

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

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

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

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

  The morphism function of tau is the tau class function for the multiplication pullback diagram (opspan).

```
(35) (KIF$function tau)
     (= (KIF$source tau) (KIF$pullback multipliable-opspan))
     (= (KIF$target tau) GPH$graph-morphism)
     (forall (?g1 (GPH$graph ?g1) ?g2 (GPH$graph ?g2) (multipliable ?g1 ?g2))
         (and (= (source (tau [?g1 ?g2]))
                 (GPH$multiplication [(GPH$opposite ?g2) (GPH$opposite ?g1)]))
              (= (target (tau [?g1 ?g2]))
                 (GPH$opposite (GPH$multiplication [?g1 ?g2])))
              (= (object (tau [?g1 ?g2]))
                 (SET.FTN$identity (GPH$object ?g1)))
              (= (morphism (tau [?g1 ?g2]))
                 (SET.LIM.PBK$tau (GPH$multiplication-opspan [?g1 ?g2])))))
```

  We can prove the following facts.

```
     (forall (?g1 (GPH$graph ?g1) ?g2 (GPH$graph ?g2) (multipliable ?g1 ?g2))
         (and (isomorphism (tau [?g1 ?g2]))
              (2-cell (tau [?g1 ?g2])))))
              (isomorphic
                  (GPH$opposite (GPH$multiplication [?g1 ?g2]))
                  (GPH$multiplication [(GPH$opposite ?g2) (GPH$opposite ?g1)]))))
```

Coherence

Associative Law

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

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

is illustrated in Figure 9.

$$\begin{tikzcd}
 & mor(G_0)\times_{obj}(mor(G_1)\times_{obj}mor(G_2)) \arrow[rrrr, shift left, "mor(\alpha_{G0,\,G1,\,G2})"] \arrow[dd, "1st_{01}'"'] \arrow[dr, "media-"] \arrow[drrr, "2nd_{01}'"] & & & & (mor(G_0)\times_{obj}mor(G_1))\times_{obj}mor(G_2 \arrow[llll, shift left, "mor(\alpha'_{G0,\,G1,\,G2})"] \arrow[dlll, "1st_{12}'"'] \arrow[dd, "2nd_{12}'"] & \\
 & & mor(G_0)\times_{obj}mor(G_1 \arrow[dl, "1st_{01}"'] \arrow[dr, "2nd_{01}"] & & mor(G_1)\times_{obj}mor(G_2 \arrow[dl, "1st_{12}"'] \arrow[dr, "2nd_{12}"] & & \\
 & mor(G_0) \arrow[dl, "src(G_0)"'] \arrow[dr, "tgt(G_0)"] & & mor(G_1) \arrow[dl, "src(G_1)"'] \arrow[dr, "tgt(G_1)"] & & mor(G_2) \arrow[dl, "src(G_2)"'] \arrow[dr, "tgt(G_2)"] & \\
obj & & obj & & obj & & obj
\end{tikzcd}$$

Figure 9: Associativity Graph Isomorphism

In order to define the morphism function

$$mor(\alpha_{G0,\,G1,\,G2}) : mor(\boldsymbol{G}_0)\times_{obj}(mor(\boldsymbol{G}_1)\times_{obj}mor(\boldsymbol{G}_2)) \rightarrow (mor(\boldsymbol{G}_0)\times_{obj}mor(\boldsymbol{G}_1))\times_{obj}mor(\boldsymbol{G}_2),$$

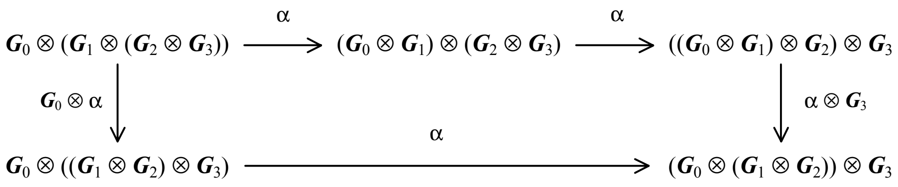
we need to specify the following auxiliary components for the associative law.

- The cone *first-cone*, which consists of
  - vertex: $mor(\boldsymbol{G}_0)\times_{obj}(mor(\boldsymbol{G}_1)\times_{obj}mor(\boldsymbol{G}_2))$,
  - first function: $1st_{01}' : mor(\boldsymbol{G}_0)\times_{obj}(mor(\boldsymbol{G}_1)\times_{obj}mor(\boldsymbol{G}_2)) \rightarrow mor(\boldsymbol{G}_0)$,
  - second function: $2nd_{01}' \cdot 1st_{12} : mor(\boldsymbol{G}_0)\times_{obj}(mor(\boldsymbol{G}_1)\times_{obj}mor(\boldsymbol{G}_2)) \rightarrow mor(\boldsymbol{G}_1)$, and
  - opspan: opspan of the multiplication $\boldsymbol{G}_0 \otimes \boldsymbol{G}_1$.
- The opspan *opspan12-3*, which consists of
  - opvertex: *obj*,
  - opfirst function: $2nd_{01}\cdot\, tgt(\boldsymbol{G}_1) : mor(\boldsymbol{G}_0)\times_{obj}mor(\boldsymbol{G}_1) \rightarrow obj$, and
  - opsecond function: $src(\boldsymbol{G}_2) : mor(\boldsymbol{G}_2) \rightarrow obj$.
- The cone *second-cone*, which consists of
  - vertex: $mor(\boldsymbol{G}_0)\times_{obj}(mor(\boldsymbol{G}_1)\times_{obj}mor(\boldsymbol{G}_2))$,
  - first function: $mediator_{01} : mor(\boldsymbol{G}_0)\times_{obj}(mor(\boldsymbol{G}_1)\times_{obj}mor(\boldsymbol{G}_2)) \rightarrow mor(\boldsymbol{G}_0)\times_{obj}mor(\boldsymbol{G}_1)$ of *first-cone*,
  - second function: $2nd_{01}' \cdot 2nd_{12} : mor(\boldsymbol{G}_0)\times_{obj}(mor(\boldsymbol{G}_1)\times_{obj}mor(\boldsymbol{G}_2)) \rightarrow mor(\boldsymbol{G}_2)$, and
  - opspan: *opspan12-3*.

The morphism function $mor(\alpha_{G0,\,G1,\,G2})$ is the mediator function of the pullback cone *second-cone*. For convenience of reference, this morphism is called the *associativity* morphism.

```
(36) (KIF$partial-function first-cone)
     (= (KIF$source first-cone) (KIF$power [KIF$three GPH$graph]))
     (= (KIF$target first-cone) SET.LIM.PBK$cone)
     (forall (?g0 (GPH$graph ?g0) ?g1 (GPH$graph ?g1) ?g2 (GPH$graph ?g2))
         (<=> ((KIF$domain first-cone) [?g0 ?g1 ?g2])
              (and (multipliable ?g0 ?g1) (multipliable ?g1 ?g2))))
     (forall (?g0 (GPH$graph ?g0) ?g1 (GPH$graph ?g1)
              ?g2 (GPH$graph ?g2) ((KIF$domain first-cone) [?g0 ?g1 ?g2]))
         (and (= (SET.LIM.PBK$vertex (first-cone [?g0 ?g1 ?g2]))
                 (SET.LIM.PBK$pullback
                     (GPH$multiplication-opspan [?g0 (GPH$multiplication ?g1 ?g2)])))
              (= (SET.LIM.PBK$first (first-cone [?g0 ?g1 ?g2]))
                 (SET.LIM.PBK$projection1
                     (GPH$multiplication-opspan [?g0 (GPH$multiplication ?g1 ?g2)])))
              (= (SET.LIM.PBK$second (first-cone [?g0 ?g1 ?g2]))
                 (SET.FTN$composition
                     [(SET.LIM.PBK$projection2
                         (GPH$multiplication-opspan [?g0 (GPH$multiplication ?g1 ?g2)]))
                      (SET.LIM.PBK$projection1 (GPH$multiplication-opspan [?g1 ?g2]))]))
              (= (SET.LIM.PBK$cone-diagram (first-cone [?g0 ?g1 ?g2]))
                 (GPH$multiplication-opspan [?g0 ?g1]))))

(37) (KIF$partial-function opspan12-3)
     (= (KIF$source opspan12-3) (KIF$power [KIF$three GPH$graph]))
     (= (KIF$target opspan12-3) SET.LIM.PBK$opspan)
     (forall (?g0 (GPH$graph ?g0) ?g1 (GPH$graph ?g1) ?g2 (GPH$graph ?g2))
         (<=> ((KIF$domain opspan12-3) [?g0 ?g1 ?g2])
              (and (multipliable ?g0 ?g1) (multipliable ?g1 ?g2))))
     (forall (?g0 (GPH$graph ?g0) ?g1 (GPH$graph ?g1)
              ?g2 (GPH$graph ?g2) ((KIF$domain opspan12-3) [?g0 ?g1 ?g2]))
         (and (= (SET$opvertex (opspan12-3 [?g0 ?g1 ?g2]))
                 (GPH$object ?g1))
              (= (SET$opfirst (opspan12-3 [?g0 ?g1 ?g2]))
                 (SET.FTN$composition
                     [(SET.LIM.PBK$projection2 (GPH$multiplication-opspan [?g0 ?g1]))
                      (target ?g1)]))
              (= (SET$opsecond (opspan12-3 [?g0 ?g1 ?g2]))
                 (source ?g2))))

(38) (KIF$partial-function second-cone)
     (= (KIF$source second-cone) (KIF$power [KIF$three GPH$graph]))
     (= (KIF$target second-cone) SET.LIM.PBK$cone)
     (forall (?g0 (GPH$graph ?g0) ?g1 (GPH$graph ?g1) ?g2 (GPH$graph ?g2))
         (<=> ((KIF$domain second-cone) [?g0 ?g1 ?g2])
              (and (multipliable ?g0 ?g1) (multipliable ?g1 ?g2))))
     (forall (?g0 (GPH$graph ?g0) ?g1 (GPH$graph ?g1)
              ?g2 (GPH$graph ?g2) ((KIF$domain second-cone) [?g0 ?g1 ?g2]))
         (and (= (SET$vertex (second-cone [?g0 ?g1 ?g2]))
                 (SET.LIM.PBK$pullback
                     (GPH$multiplication-opspan [?g0 (GPH$multiplication [?g1 ?g2])])))
              (= (SET.LIM.PBK$first (second-cone [?g0 ?g1 ?g2]))
                 (SET.LIM.PBK$mediator (first-cone [?g0 ?g1 ?g2])))
              (= (SET.LIM.PBK$second (second-cone [?g0 ?g1 ?g2]))
                 (SET.FTN$composition
                     [(SET.LIM.PBK$projection2
                        (GPH$multiplication-opspan [?g0 (GPH$multiplication [?g1 ?g2])]))
                      (SET.LIM.PBK$projection2 (GPH$multiplication-opspan [?g1 ?g2]))]))
              (= (SET.LIM.PBK$cone-diagram (second-cone [?g0 ?g1 ?g2]))
                 (opspan12-3 [?g0 ?g1 ?g2]))))

(39) (KIF$partial-function alpha)
     (= (KIF$source alpha) (KIF$power [KIF$three GPH$graph]))
     (= (KIF$target alpha) graph-morphism)
     (forall (?g0 (GPH$graph ?g0) ?g1 (GPH$graph ?g1) ?g2 (GPH$graph ?g2))
         (<=> ((KIF$domain alpha) [?g0 ?g1 ?g2])
              (and (multipliable ?g0 ?g1) (multipliable ?g1 ?g2))))
     (forall (?g0 (GPH$graph ?g0) ?g1 (GPH$graph ?g1)
              ?g2 (GPH$graph ?g2) ((KIF$domain alpha) [?g0 ?g1 ?g2]))
         (and (= (source (alpha [?g0 ?g1 ?g2]))
                 (GPH$multiplication [?g0 (GPH$multiplication [?g1 ?g2])]))
              (= (target (alpha [?g0 ?g1 ?g2]))
                 (GPH$multiplication [(GPH$multiplication [?g1 ?g0]) ?g2]))
              (= (object (alpha [?g0 ?g1 ?g2]))
                 (SET.FTN$identity (GPH$object ?g0)))
              (= (morphism (alpha [?g0 ?g1 ?g2]))
                 (SET.LIM.PBK$mediator (second-cone [?g0 ?g1 ?g2])))))

(40) (KIF$partial-function associativity)
     (= (KIF$source associativity) (KIF$power [KIF$three GPH$graph]))
     (= (KIF$target associativity) SET.FTN$function)
     (forall (?g0 (GPH$graph ?g0) ?g1 (GPH$graph ?g1) ?g2 (GPH$graph ?g2))
         (<=> ((KIF$domain associativity) [?g0 ?g1 ?g2])
              (and (multipliable ?g0 ?g1) (multipliable ?g1 ?g2))))
     (forall (?g0 (GPH$graph ?g0) ?g1 (GPH$graph ?g1)
              ?g2 (GPH$graph ?g2) ((KIF$domain associativity) [?g0 ?g1 ?g2]))
         (= (associativity [?g0 ?g1 ?g2]) (morphism (alpha [?g0 ?g1 ?g2]))))
```

The oppositely directed graph morphism $\alpha'_{\boldsymbol{G}0,\,\boldsymbol{G}1,\,\boldsymbol{G}2} : (\boldsymbol{G}_0 \otimes \boldsymbol{G}_1) \otimes \boldsymbol{G}_2 \rightarrow \boldsymbol{G}_0 \otimes (\boldsymbol{G}_1 \otimes \boldsymbol{G}_2)$ can be defined in a similar fashion, and, based upon uniqueness of the pullback mediator function, the two can be shown to be inverses. In addition, the associative coherence theorem in Diagram 2 can be proven.



$$\begin{tikzcd}
\boldsymbol{G}_0 \otimes (\boldsymbol{G}_1 \otimes (\boldsymbol{G}_2 \otimes \boldsymbol{G}_3)) \arrow[r, "\alpha"] \arrow[d, "\boldsymbol{G}_0 \otimes \alpha"'] & (\boldsymbol{G}_0 \otimes \boldsymbol{G}_1) \otimes (\boldsymbol{G}_2 \otimes \boldsymbol{G}_3) \arrow[r, "\alpha"] & ((\boldsymbol{G}_0 \otimes \boldsymbol{G}_1) \otimes \boldsymbol{G}_2) \otimes \boldsymbol{G}_3 \arrow[d, "\alpha \otimes \boldsymbol{G}_3"] \\
\boldsymbol{G}_0 \otimes ((\boldsymbol{G}_1 \otimes \boldsymbol{G}_2) \otimes \boldsymbol{G}_3) \arrow[rr, "\alpha"] & & (\boldsymbol{G}_0 \otimes (\boldsymbol{G}_1 \otimes \boldsymbol{G}_2)) \otimes \boldsymbol{G}_3
\end{tikzcd}$$

Diagram 2: Associativity Coherence

Unit Laws

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

$$\begin{tikzcd}
 & obj(\boldsymbol{G})\times_{obj(\boldsymbol{G})}mor(\boldsymbol{G}) \arrow[dl, "1st"'] \arrow[dr, bend left, "mor(\lambda_{\boldsymbol{G}}) = 2nd"] & & \\
obj(\boldsymbol{G}) \arrow[d, "id_{obj(\boldsymbol{G})}"'] \arrow[dr, "id_{obj(\boldsymbol{G})}"] & & mor(\boldsymbol{G}) \arrow[ul, bend left, "mor(\lambda'_{\boldsymbol{G}})"] \arrow[dl, "src(\boldsymbol{G})"'] \arrow[d, "tgt(\boldsymbol{G})"] & \\
obj(\boldsymbol{G}) & obj(\boldsymbol{G}) & obj(\boldsymbol{G}) &
\end{tikzcd}
\qquad
\begin{tikzcd}
 & mor(\boldsymbol{G})\times_{obj(\boldsymbol{G})}obj(\boldsymbol{G}) \arrow[dl, bend right, "mor(\rho_{\boldsymbol{G}}) = 1st"'] \arrow[dr, "2nd"] & \\
mor(\boldsymbol{G}) \arrow[ur, bend right, "mor(\rho'_{\boldsymbol{G}})"'] \arrow[d, "src(\boldsymbol{G})"'] \arrow[dr, "tgt(\boldsymbol{G})"] & & obj(\boldsymbol{G}) \arrow[dl, "id_{obj(\boldsymbol{G})}"'] \arrow[d, "id_{obj(\boldsymbol{G})}"] \\
obj(\boldsymbol{G}) & obj(\boldsymbol{G}) & obj(\boldsymbol{G})
\end{tikzcd}$$

Figure 8: Left Unit and Right Unit Graph Isomorphism

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

(42) (KIF$function right)
     (= (KIF$source right) GPH$graph)
     (= (KIF$target right) graph-morphism)
     (forall (?g (GPH$graph ?g))
         (and (= (source (right ?g))
                 (GPH$multiplication [?g (GPH$unit (GPH$object ?g))]))
              (= (target (right ?g)) ?g)
              (= (object (right ?g))
                 (SET.FTN$identity (GPH$object ?g)))
              (= (morphism (right ?g))
                 (SET.LIM.PBK$projection1
                     (GPH$multiplication-opspan [(GPH$unit (GPH$object ?g)) ?g])))))
```

An oppositely directed graph morphism $\lambda' : \boldsymbol{G} \rightarrow 1_{obj(\boldsymbol{G})} \otimes \boldsymbol{G}$ can be defined, whose morphism function $mor(\lambda') : mor(\boldsymbol{G}) \rightarrow obj(\boldsymbol{G})\times_{obj(\boldsymbol{G})}mor(\boldsymbol{G})$ is the mediator function for a pullback cone over the opspan associated with $1_{obj(\boldsymbol{G})} \otimes \boldsymbol{G}$, whose vertex is $mor(\boldsymbol{G})$, whose first function is $src(\boldsymbol{G})$ and whose second function is $id_{mor(\boldsymbol{G})}$. Based upon uniqueness of the pullback mediator function, this can be shown to be inverse to λ. Similarly, an oppositely directed graph morphism $\rho' : \boldsymbol{G} \rightarrow \boldsymbol{G} \otimes 1_{obj(\boldsymbol{G})}$ can be defined and shown to be the inverse to ρ. In addition, the unit coherence theorem and identity theorem in Diagram 3 can be proven.

$$\begin{tikzcd}
\boldsymbol{G}_0 \otimes (1_C \otimes \boldsymbol{G}_2)) \arrow[r, "\alpha"] \arrow[d, "\boldsymbol{G}_0 \otimes \lambda"'] & (\boldsymbol{G}_0 \otimes 1_C) \otimes \boldsymbol{G}_2 \arrow[d, "\rho \otimes \boldsymbol{G}_2"] \\
\boldsymbol{G}_0 \otimes \boldsymbol{G}_2 \arrow[r, phantom, "="] & \boldsymbol{G}_0 \otimes \boldsymbol{G}_2
\end{tikzcd}
\qquad
\lambda_1 = \rho_1 : 1_C \otimes 1_C \longrightarrow 1_C$$

Diagram 3: Unit Coherence