The Namespace of Sets

This namespace axiomatizes Set the category of sets and functions. This is the most basic category in the IFF Lower Core (meta) Ontology (IFF-LCO).

Things still to do:
- limits of diagram morphisms; products of tuple morphisms
- binary product morphisms; equalizer morphisms; pullback morphisms
- (co)kernels and other special (co)limits

Table 1 lists the basic terminology in the namespace of sets.

Table 1: Basic Terminology for the Namespace of Sets

	Category	Functor	Natural Transformation	Adjunction
set	set	(semipair ?x) type-power		

	Class	Function	Other
set.obj	object = set		subset disjoint (member ?a)
		power	
		type-power semipair	
set.mor	morphism = function injection surjection bijection	source target composition identity inverse constant inclusion inverse-image	composable-opspan composable
		union intersection tuple-union tuple-intersection partition	
		semipair type-power	

Table 2 lists the diagram terminology in the namespace of sets.

Table 2: Diagram Terminology for the Namespace of Sets

	Collection/Class	Function	Other
set.dgm	object = diagram discrete	graph = shape set function	mixed-composable-opspan mixed-composable
		tuple	
		diagram-fiber set-fiber function-fiber	
		inverse-image mixed-composition	
set.dgm .mor	morphism	source target graph = shape component composition identity	composable-opspan composable
		tuple	
		morphism-fiber source-fiber target-fiber component-fiber	
		inverse-image mixed-composition	mixed-composable-opspan mixed-composable
set.dgm .lmor	lax-morphism	source target graph-morphism = shape composition identity	composable-opspan composable
		target-diagram diagram-morphism	
set.dgm .clmor	colax-morphism	source target graph-morphism = shape composition identity	composable-opspan composable
		target-diagram diagram-morphism	
set.dgm .tpl	tuple = object	index set diagram	
		tuple-fiber inclusion set-fiber	
		inverse-image mixed-composition	mixed-composable-opspan mixed-composable
set.dgm .tpl.mor	morphism	source target index component diagram composition identity	composable-opspan composable
		inverse-image mixed-composition	mixed-composable-opspan mixed-composable
set.dgm .tpl.lmor	lax-morphism	source target set-function target-tuple tuple-morphism composition identity	composable-opspan composable
set.dgm .tpl.clmor	colax-morphism	source target set-function target-tuple tuple-morphism composition identity	composable-opspan composable
set.dgm .art	arity = object	index base set	
		arity-fiber inclusion base-fiber set-fiber	
		inverse-image mixed-composition	mixed-composable-opspan mixed-composable
set.dgm .art.mor	morphism	source target index base quartet composition identity	composable-opspan composable

set.dgm .pr	diagram = pair	set1 set2	
set.dgm .pr.mor	morphism	source target function1 function2	
set.dgm .ppr	diagram = parallel-pair	origin destination function1 function2	
		endorelation subobject	
set.dgm .ppr.mor	morphism	source target origin destination	
set.dgm .spn	diagram = span	set1 set2 vertex first second	
		opposite	
		pair parallel-pair standard-parallel-pair	
set.dgm .spn.mor	morphism	source target function1 function2 vertex	
set.dgm .ospn	diagram = opspan	set1 set2 opvertex opfirst opsecond	
		opposite	
		pair parallel-pair standard-parallel-pair	
set.dgm .ospn.mor	morphism	source target function1 function2 opvertex	

Table 3 lists the limit terminology in the namespace of sets.

Table 3: Limit Terminology for the Namespace of Sets

	Collection/Class	Function	Other
set.lim	cone	cone-diagram vertex component	terminal unique
		limiting-cone limit projection mediator	
		tuple target-tuple target-cone target-function source-image-cone source-image-function parallel-pair standard-limiting-cone standard-limit standard-projection	
		diagram-shape cone-fiber cone-diagram-fiber vertex-fiber component-fiber limiting-cone-fiber limit-fiber projection-fiber mediator-fiber	
set.lim .mor		morphism-cone limit connection-cone connection lax-limit	
set.lim .prd	cone	cone-tuple vertex component	
		limiting-cone limit projection mediator tupling-cone tupling	
		standard-limiting-cone standard-limit = standard-product = cartesian-product standard-projection	
set.lim .prd2	cone	cone-diagram vertex first second	
		limiting-cone limit = binary-product projection1 projection2 mediator	
		standard-limiting-cone standard-limit = standard-binary-product = cartesian-binary-product standard-projection1 standard-projection2	
set.lim .sub	subset	set = base element range canon subobject mediator standard-canon standard-subobject	contains matches-opspan matches respects
set.lim .equ	cone	cone-diagram vertex function	
		limiting-cone limit = equalizer canon mediator	
		standard-limiting-cone standard-limit = standard-equalizer standard-canon	
set.lim .pbk	cone	cone-diagram vertex first second binary-product-cone equalizer-cone	
		limiting-cone limit = pullback projection1 projection2 mediator	
		standard-limiting-cone standard-limit = standard-pullback standard-projection1 standard-projection2	

Table 4 lists the colimit terminology in the namespace of sets.

Table 4: Colimit Terminology for the Namespace of Sets

	Collection/Class	Function	Other
set.col	cocone	cocone-diagram opvertex component	initial counique
		colimiting-cocone colimit injection comediator	
		tuple source-tuple source-cone source-function image-target-cocone image-target-function parallel-pair standard-colimiting-cocone standard-colimit standard-injection	
		diagram-shape cocone-fiber cocone-diagram-fiber opvertex-fiber component-fiber colimiting-cocone-fiber colimit-fiber injection-fiber comediator-fiber	
set.col .mor		morphism-cocone colimit connection-cocone connection colax-colimit	
set.col .coprd	cocone	cocone-tuple opvertex component	
		colimiting-cocone colimit = coproduct injection	
		standard-colimiting-cocone standard-colimit = standard-coproduct = disjoint-union standard-injection comediator cotupling-cocone cotupling constant-index indication	
		tuple-index cocone-fiber cocone-tuple-fiber opvertex-fiber component-fiber colimiting-cocone-fiber colimit-fiber = coproduct-fiber injection-fiber comediator-fiber	
set.col .coprd.mor		morphism-cocone coproduct connection-cocone connection colax-colimit indication	
set.col .art	cocone	cocone-arity opvertex component	
		colimiting-cocone colimit = coproduct injection	
		standard-colimiting-cocone standard-colimit = standard-coproduct = disjoint-union standard-injection comediator cotupling-cocone cotupling constant-index indication inclusion projection standard-indication standard-projection	
set.col .art.mor		base-restriction coproduct-tuple coproduct indication projection	
set.col .coprd2	cocone	cocone-diagram opvertex opfirst opsecond	
		colimiting-cocone	

		colimit = binary-coproduct injection1 injection2 comediator	
		standard-colimiting-cocone standard-colimit = standard-binary-coproduct = disjoint-union standard-injection1 standard-injection2	
set.col .coprd2.mor		morphism-cocone morphism	
set.col .endo	endorelation	set = base element kernel canon quotient comediator standard-canon standard-quotient	subrelation matches-opspan matches respects
set.col .coeq	cocone	cocone-diagram opvertex function	
		colimiting-cocone colimit = coequalizer canon comediator	
		standard-colimiting-cocone standard-colimit = standard-coequalizer standard-canon	
set.col .coeq.mor		morphism-cocone morphism	
set.col .psh	cocone	cocone-diagram opvertex opfirst opsecond binary-coproduct-cocone coequalizer-cocone	
		colimiting-cocone colimit = pushout injection1 injection2 comediator	
		standard-colimiting-cocone standard-colimit = standard-pushout standard-injection1 standard-injection2	
set.col .psh.mor		morphism-cocone morphism	

The Context of Sets

`set`

Categories

The most basic (and useful) category specified within the IFF Lower Core (meta) Ontology (IFF-LCO) is Set, the category of sets and functions. This category is axiomatized in the set namespace of the IFF-LCO.

The category of sets serves as the target category for many functors and natural transformations of the IFF-LCO. Additionally, the limit/colimit constructions in the category Set provide concrete representations for the limit/colimit constructions in other categories of the IFF-LCO.

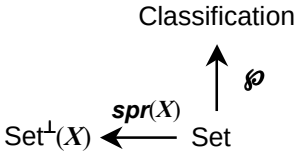


Figure 1: Set Functors

```
(1) (CAT$category set)
```

Functors

Here is a list of the inner shell functors that are axiomatized in this namespace (Figure 1).

- ***semipair***$(\boldsymbol{X}) : \mathsf{Set} \to \mathsf{Set}^{\perp}(\boldsymbol{X})$: For any set $\boldsymbol{X}$ representing a fixed set of names, there is a fiber *semipair* functor from Set to the fiber semipair category $\mathsf{Set}^{\perp}(\boldsymbol{X})$. It maps a set A to the set pair whose index set is $\boldsymbol{X}$ and whose object set is A. It maps a function $f\colon A \to B$ to the semiquartet whose index bijection is $id_{\boldsymbol{X}}\colon \boldsymbol{X} \to \boldsymbol{X}$ and whose object function is $f\colon A \to B$. This is one half of an inverse pair of functors, which establish the isomorphism $\mathsf{Set} \cong \mathsf{Set}^{\perp}(\boldsymbol{X})$.
- $\wp : \mathsf{Set} \to \mathsf{Classification}$: There is a (*type*) *power* functor from Set to Classification the category of classifications and infomorphisms. It maps a set A to the type power classification $\wp(A) = \langle \wp A, A, (\in_{\wp A})^{\mathrm{op}} \rangle$. It maps a function $f\colon A \to B$ to the infomorphism $\wp(f) = \langle f^{-1}, f \rangle\colon \wp(A) \rightleftarrows \wp(B)$ whose type function is $f\colon A \to B$ and whose instance function is inverse-image $f^{-1}\colon \wp B \to \wp A$. This is the left adjoint of an adjunction, whose right adjoint is the underlying type functor.

```
(2) (KIF$function semipair)
    (= (KIF$source semipair) set.obj$object)
    (= (KIF$target semipair) FUNC$functor)
    (forall (?x (set.obj$object ?x))
        (and (= (FUNC$source (semipair ?x)) set)
             (= (FUNC$target (semipair ?x)) (spr.fbr$semipair ?x))
             (= (FUNC$object (semipair ?x)) (spr.obj$semipair ?x))
             (= (FUNC$morphism (semipair ?x)) (spr.mor$semipair ?x))))

(3) (FUNC$functor type-power)
    (= (FUNC$source type-power) set)
    (= (FUNC$target type-power) cls$classification)
    (= (FUNC$object type-power) set.obj$power)
    (= (FUNC$morphism type-power) set.mor$power)
```

Sets

`set.obj`

A *set* A is a small collection. The class of all sets is a subcollection of the collection of all classes.

```
(1) (SET$class object)
    (SET$class set)
    (= set object)
    (KIF$subcollection object SET$class)
```

There is a *subset* relation $\subseteq$ on sets. A set A is a subset of a set B, $A \subseteq B$, when every element of A is an element of B. The subset relation is an abridgment of the subclass relation.

```
(2) (REL$relation subset)
    (= (REL$class1 subset) set.obj$object)
    (= (REL$class2 subset) set.obj$object)
    (REL$abridgement subset SET$subclass)
```

A *disjoint* relation restricts the class disjoint relation to sets.

```
(3) (REL$relation disjoint)
    (= (REL$class1 disjoint) object)
    (= (REL$class2 disjoint) object)
    (REL$abridgement disjoint SET$disjoint)
```

For any set A, there is a *power* function $\wp$: set $\to$ set that maps any A to its power set $\wp A$, the set of all subsets of A. Set power is a restriction of class power.

```
(4) (SET.FTN$function power)
    (= (SET.FTN$source power) set.obj$object)
    (= (SET.FTN$target power) set.obj$object)
    (SET.FTN$restriction power SET$power)
```

For any set A there is a *member*ship relation $\in_A = (A, \wp A)$, whose first set is A and whose second set is the power set of A.

```
(5) (SET.FTN$function member)
    (= (SET.FTN$source member) object)
    (= (SET.FTN$target member) rel$relation)
    (= (SET.FTN$composition [member rel$first]) (SET.FTN$identity set.obj$object))
    (= (SET.FTN$composition [member rel$second]) power)
    (forall (?a (object ?a) ?x (?a ?x) ?b (subset ?b ?a))
        (<=> ((member ?a) ?x ?b) (?b ?x)))
```

For any set A there is a *type power* classification $\wp A = (\wp A, A, \in^{\mathrm{op}}_A)$, whose instance set is the power set of A, whose type set is A and whose incidence is the opposite of membership on A.

```
(6) (SET.FTN$function type-power)
    (= (SET.FTN$source type-power) object)
    (= (SET.FTN$target type-power) cls.obj$classification)
    (= (SET.FTN$composition [type-power cls.obj$instance]) power)
    (= (SET.FTN$composition [type-power cls.obj$type]) (SET.FTN$identity set.obj$object))
    (= type-power (SET.FTN$composition [member rel$opposite]))
```

Any set $\boldsymbol{X}$ defines a *semipair* function from the class of sets to the class of semipairs with index $\boldsymbol{X}$

$\mathit{spr}(\boldsymbol{X})$: set $\to$ spr($\boldsymbol{X}$).

For any set $\boldsymbol{X}$, the composition of $\mathit{spr}(\boldsymbol{X})$ with $\mathit{set}(\boldsymbol{X})$ is the identity on set. Hence, the class functions $\mathit{spr}(\boldsymbol{X})$ with $\mathit{set}(\boldsymbol{X})$ are inverses. This helps establish the isomorphism

$\mathrm{Set}^{\perp}(\boldsymbol{X}) \cong$ Set.

```
(7) (KIF$function semipair)
    (= (KIF$source semipair) object)
    (= (KIF$target semipair) (SET.FTN$function)
    (forall (?x (object ?x))
        (and (= (SET.FTN$source (semipair ?x)) object)
             (= (SET.FTN$target (semipair ?x)) (spr.fbr.obj$object ?x))
             (= (SET.FTN$composition [(semipair ?x) (spr.fbr.obj$index ?x)])
                ((SET.FTN$constant [object (spr.fbr.obj$object ?x)]) ?x))
```

```
(= (SET.FTN$composition [(semipair ?h) (spr.fbr.obj$set ?h)])
   (SET.FTN$identity object))))
```

Functions

`set.mor`

Sets are related through set functions. A set function $f : A \to B$ has source set A and target set B. The class of all set functions is a subcollection of the collection of all class functions.

```
(1) (SET$class morphism)
    (SET$class function)
    (KIF$subcollection morphism SET.FTN$function)

(2) (SET.FTN$function source)
    (= (SET.FTN$source source) morphism)
    (= (SET.FTN$target source) set.obj$object)
    (KIF$restriction source SET.FTN$source)

(3) (SET.FTN$function target)
    (= (SET.FTN$source target) morphism)
    (= (SET.FTN$target target) set.obj$object)
    (KIF$restriction target SET.FTN$target)
```

A function $f : A \to B$ is an *injection* when it is one-one. A function $f : A \to B$ is a *surjection* when it onto. A function $f : A \to B$ is a *bijection* when it is a one-one correspondence; that is, when it is both a surjection and an injection. The class of all set (injections, surjections) bijections is a subcollection of the collection of all class (injections, surjections) bijections.

```
(4) (SET$class injection)
    (SET$subclass injection morphism)
    (= injection (SET$binary-intersection [morphism SET.FTN$injection]))

(5) (SET$class surjection)
    (SET$subclass surjection morphism)
    (= surjection (SET$binary-intersection [morphism SET.FTN$surjection]))

(6) (SET$class bijection)
    (SET$subclass bijection morphism)
    (= bijection (SET$binary-intersection [morphism SET.FTN$bijection]))
```

For any two sets A and B and any element $y \in B$ there is a *constant y* function from A to B.

```
(7) (SET.FTN$function constant)
    (= (SET.FTN$source constant) (SET.LIM.PRD2$binary-product set.obj$object)
    (= (SET.FTN$target constant) SET.FTN$function)
    (forall (?a (set.obj$object ?a) ?b (set.obj$object ?b))
        (and (= (SET.FTN$source (constant [?a ?b])) ?b)
             (= (SET.FTN$target (constant [?a ?b])) morphism)
             (= (SET.FTN$composition [(constant [?a ?b]) source]) ?a)
             (= (SET.FTN$composition [(constant [?a ?b]) target]) ?b)
             (SET.FTN$restriction (constant [?a ?b]) (SET.FTN$constant [?a ?b])))))
```

For any two sets that are ordered by inclusion $A \subseteq B$ there is an *inclusion* function $A \to B$.

```
(8) (SET.FTN$function inclusion)
    (= (SET.FTN$source inclusion) (REL$extent set.obj$subset))
    (= (SET.FTN$target inclusion) morphism)
    (= (SET.FTN$composition [inclusion source]) (REL$first set.obj$subset))
    (= (SET.FTN$composition [inclusion source]) (REL$second set.obj$subset))
    (SET.FTN$restriction inclusion SET.FTN$inclusion)
```

Set functions are *composable* when the target of the first is equal to the source of the second. The *composition* of two composable functions is defined pointwise.

```
 (9) (SET.LIM.PBK$opspan composable-opspan)
     (= (SET.LIM.PBK$class1 composable-opspan) morphism)
     (= (SET.LIM.PBK$class2 composable-opspan) morphism)
     (= (SET.LIM.PBK$opvertex composable-opspan) set.obj$object)
     (= (SET.LIM.PBK$first composable-opspan) target)
     (= (SET.LIM.PBK$second composable-opspan) source)
     (KIF$subcollection
```

```
        (SET.LIM.PBK$pullback composable-opspan)
        (KIF$pullback SET.FTN$composable-opspan))

(10) (REL$relation composable)
     (= (REL$class1 composable) morphism)
     (= (REL$class2 composable) morphism)
     (= (REL$extent composable) (SET.LIM.PBK$pullback composable-opspan))
     (REL$subrelation composable SET.FTN$composable)

(11) (SET.FTN$function composition)
     (= (SET.FTN$source composition) (REL$extent composable))
     (= (SET.FTN$target composition) morphism)
     (KIF$restriction composition SET.FTN$composition)
```

Composition satisfies the usual *associative law*.

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

For any set, there is an *identity* function.

```
(13) (SET.FTN$function identity)
     (= (SET.FTN$source identity) set.obj$object)
     (= (SET.FTN$target identity) morphism)
     (= (SET.FTN$composition [identity source]) (SET.FTN$identity set.obj$object))
     (= (SET.FTN$composition [identity target]) (SET.FTN$identity set.obj$object))
     (SET.FTN$restriction identity SET.FTN$identity)
```

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

```
(14) (forall (?f (morphism ?f))
         (and (= (composition [(identity (source ?f)) ?f]) ?f)
              (= (composition [?f (identity (target ?f))]) ?f)))
```

There is an inverse class function on the class of bijections.

```
(15) (SET.FTN$function inverse)
     (= (SET.FTN$source inverse) bijection)
     (= (SET.FTN$target inverse) bijection)
     (= (SET.FTN$composition [inverse source]) target)
     (= (SET.FTN$composition [inverse target]) source)
     (forall (?f (bijection ?f))
          (and (= (composition [?f (inverse ?f)]) (identity (source ?f)))
               (= (composition [(inverse ?f) ?f]) (identity (target ?f)))))
```

Following the assumption that the power of a set is a set, we also assume that the power of a set function is a set function. This takes two forms: the direct image and the inverse image. For any set function $f : A \to B$ there is an *inverse image* function $f^{-1} : \wp B \to \wp A$ defined by $f^{-1}(Y) = \{x \in A \mid f(x) \in Y\} \subseteq A$ for any subset $Y \subseteq B$.

```
(16) (SET.FTN$function inverse-image)
     (= (SET.FTN$source inverse-image) function)
     (= (SET.FTN$target inverse-image) function)
     (= (SET.FTN$composition [inverse-image source])
        (SET.FTN$composition [target SET$power]))
     (= (SET.FTN$composition [inverse-image target])
        (SET.FTN$composition [source SET$power]))
     (SET.FTN$restriction inverse-image SET.FTN$inverse-image)
```

For any function $f : A \to B$ there is a *type power* infomorphism $\wp f = (f^{-1}, f) : \wp A \rightleftarrows \wp B$, whose instance function is the inverse image function of f and whose type function is f.

```
(17) (SET.FTN$function type-power)
     (= (SET.FTN$source type-power) morphism)
     (= (SET.FTN$target type-power) cls.mor$infomorphism)
     (= (SET.FTN$composition [type-power cls.mor$source])
        (SET.FTN$composition [source set.obj$type-power]))
     (= (SET.FTN$composition [type-power cls.mor$target])
        (SET.FTN$composition [target set.obj$type-power]))
     (= (SET.FTN$composition [type-power cls.mor$instance]) inverse-image)
```

```
(= (SET.FTN$composition [type-power cls.mor$type]) (SET.FTN$identity set.mor$morphism))
```

Any set bijection $\boldsymbol{h} : \boldsymbol{X} \to \boldsymbol{Y}$ defines a *semipair* function from the class of set functions to the class of semiquartets with index $\boldsymbol{h}$

$\mathit{spr}(\boldsymbol{h}) : \mathsf{ftn} \to \mathsf{sprmor}(\boldsymbol{h}) = \mathsf{sqtt}(\boldsymbol{h})$.

The composition of $\mathit{spr}(\boldsymbol{h})$ with $\mathit{set}(\boldsymbol{h})$ is the identity on ftn. Hence, the class functions $\mathit{spr}(\boldsymbol{h})$ with $\mathit{set}(\boldsymbol{h})$ are inverses. This fact when applied to identity functions is the morphism aspect for two functors that are inverses. For any set $\boldsymbol{X}$, this in turn helps establish the isomorphism

$\mathsf{Set}^{\perp}(\boldsymbol{X}) \cong \mathsf{Set}$.

```
(18) (KIF$function semipair)
     (= (KIF$source semipair) bijection)
     (= (KIF$target semipair) set.mor$morphism)
     (forall (?h (set.mor$bijection ?h))
         (and (= (SET.FTN$source (semipair ?h)) morphism)
              (= (SET.FTN$target (semipair ?h)) (spr.fbr.mor$morphism ?h))
              (= (SET.FTN$composition [(semipair ?h) (spr.fbr.mor$source ?h)])
                 (SET.FTN$composition [source (set.obj$semipair (source ?h))]))
              (= (SET.FTN$composition [(semipair ?h) (spr.fbr.mor$target ?h)])
                 (SET.FTN$composition [target (set.obj$semipair (target ?h))]))
              (= (SET.FTN$composition [(semipair ?h) (spr.fbr.mor$index ?h)])
                 ((SET.FTN$constant [morphism (spr.fbr.mor$morphism ?h)]) ?h))
              (= (SET.FTN$composition [(semipair ?h) (spr.fbr.mor$set ?h)])
                 (SET.FTN$identity morphism))))
```

For any set A, there is a *union* operator $\cup_A : \wp\wp A \to \wp A$ and an *intersection* operator $\cap_A : \wp\wp C \to \wp C$. That is, for any collection of subsets $C \subseteq \wp A$ of a set C there is a union set $\cup_A(C)$ and an intersection set $\cap_A(C)$.

```
(19) (SET.FTN$function union)
     (= (SET.FTN$source union) set.obj$object)
     (= (SET.FTN$target union) function)
     (= (SET.FTN$composition [union source])
        (SET.FTN$composition [set.obj$power set.obj$power]))
     (= (SET.FTN$composition [union target]) set.obj$power)
     (SET.FTN$restriction union SET.FTN$union)
     (forall (?a (set.obj$object ?a)) ?C (set.obj$subset C (set$power ?a)) ?x (?a ?x))
         (<=> (((union ?a) ?C) ?x)
              (exists (?b (?C ?b)) (?b ?x))))

(20) (SET.FTN$function intersection)
     (= (SET.FTN$source intersection) set.obj$object)
     (= (SET.FTN$target intersection) function)
     (= (SET.FTN$composition [intersection source])
        (SET.FTN$composition [set.obj$power set.obj$power]))
     (= (SET.FTN$composition [intersection target]) set.obj$power)
     (SET.FTN$restriction intersection SET.FTN$intersection)
     (forall (?a (set.obj$object ?a)) ?C (set.obj$subset C (set$power ?a)) ?x (?a ?x))
         (<=> (((intersection ?a) ?C) ?x)
              (forall (?b (?C ?b)) (?b ?x))))
```

For any tuple of sets $\boldsymbol{A} : J \to \mathsf{set}$, there is a *union* operator and an *intersection* operator. That is, for any tuple of sets $\boldsymbol{A} = \{\boldsymbol{A}(n) \mid n \in J\}$, there are union and intersection sets

$\cup\{\boldsymbol{A}(n) \mid n \in J\}$ and $\cap\{\boldsymbol{A}(n) \mid n \in J\}$.

The union (intersection) of a tuple is the union (intersection) of the range of its set class function.

```
(21) (SET.FTN$function tuple-union)
     (= (SET.FTN$source tuple-union) set.dgm.tpl$tuple)
     (= (SET.FTN$target tuple-union) set.obj$object)
     (forall (?a (set.dgm.tpl$tuple ?a))
         (= (tuple-union ?a) (union (SET.FTN$range (set.dgm.tpl$set ?a)))))

(22) (SET.FTN$function tuple-intersection)
     (= (SET.FTN$source tuple-intersection) set.dgm.tpl$tuple)
     (= (SET.FTN$target tuple-intersection) set.obj$object)
     (forall (?a (set.dgm.tpl$tuple ?a))
```

```
        (= (tuple-intersection ?a) (intersection (SET.FTN$range (set.dgm.tpl$set ?a)))))
```

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

```
(23) (SET.FTN$function partition)
     (= (SET.FTN$source partition) set.obj$object)
     (= (SET.FTN$target partition) set.obj$object)
     (forall (?a (set.obj$object ?a))
         (and (set.obj$subset (partition ?a) (set$power (set$power ?a))))
             (forall (?p ((set$power (set$power ?a)) ?p))
                 (<=> ((partition ?a) ?p)
                      (and (= ((union ?a) ?p) ?a)
                           (forall (?pj (?p ?pj) ?pk (?p ?pk) (not (= ?pj ?pk)))
                               (set$disjoint ?pj ?pk)))))))
     (SET.FTN$restriction partition SET.FTN$partition)
```

Diagrams for Sets

`set.dgm`

A generic Set-valued *diagram* $D : G \to |Set|$ is essentially a large graph morphism from a (small) *shape graph* G into the large underlying graph of the category Set of sets and set functions. This graph morphism has two components: the *set* class function $set(D) : node(G) \to obj(Set)$ and the (set) *function* class function $ftn(D) : edge(G) \to mor(Set)$. Diagrams are determined by their shape, set and function class functions. Set-valued diagrams correspond adjointly to Set-valued functors. Hence, the collection of all Set-valued diagrams forms a KIF collection. The set class function of a diagram corresponds to a tuple of sets.

```
(1) (KIF$collection diagram)
    (KIF$collection object)
    (= object diagram)

(2) (KIF$function graph)
    (KIF$function shape)
    (= shape graph)
    (= (KIF$source graph) diagram)
    (= (KIF$target graph) gph.obj$object)

(3) (KIF$function set)
    (= (KIF$source set) diagram)
    (= (KIF$target set) SET.FTN$function)
    (forall (?d (diagram ?d))
        (and (= (SET.FTN$source (set ?d)) (gph.obj$node (graph ?d)))
             (= (SET.FTN$target (set ?d)) set.obj$object)))

(4) (KIF$function function)
    (= (KIF$source function) diagram)
    (= (KIF$target function) SET.FTN$function)
    (forall (?d (diagram ?d))
        (and (= (SET.FTN$source (function ?d)) (gph.obj$edge (graph ?d)))
             (= (SET.FTN$target (function ?d)) set.mor$morphism)))
             (= (SET.FTN$composition [(function ?d) set.mor$source])
                (SET.FTN$composition [(gph.obj$source (graph ?d)) (set ?d)]))
             (= (SET.FTN$composition [(function ?d) set.mor$target])
                (SET.FTN$composition [(gph.obj$target (graph ?d)) (set ?d)])))))

(5) (forall (?d1 (diagram ?d1) ?d2 (diagram ?d2))
        (=> (and (= (graph ?d1) (graph ?d2))
                 (= (set ?d1) (set ?d2)))
                 (= (function ?d1) (function ?d2)))
            (= ?d1 ?d2)))
```

Any diagram has an associated *tuple* defined by its set component. The function component is forgotten.

```
(6) (KIF$function tuple)
    (= (KIF$source tuple) diagram)
    (= (KIF$target tuple) set.dgm.tpl$tuple)
    (forall (?d (diagram ?d))
        (and (= (set.dgm.tpl$index (tuple ?d)) (gph$node (graph ?d)))
             (= (set.dgm.tpl$set (tuple ?d)) (set ?d))))
```

A diagram *D* is *discrete* when it is the diagram induced by a tuple set *A*.

```
(7) (KIF$collection discrete)
    (KIF$subcollection discrete diagram)
    (forall (?d (diagram ?d))
        (<=> (discrete ?d)
             (exists (?a (set.dgm.tpl$tuple ?a))
                 (= ?d (set.dgm.tpl$diagram ?a)))))
```

The *fiber* $dgm(G) \subseteq dgm$ for any (small) graph G is the class of all diagrams having shape G.

```
 (8) (KIF$function diagram-fiber)
     (= (KIF$source diagram-fiber) gph.obj$object))
     (= (KIF$target diagram-fiber) SET$class)
```

```
     (= diagram-fiber (KIF$fiber graph))

 (9) (KIF$function inclusion)
     (= (KIF$source inclusion) gph.obj$object))
     (= (KIF$target inclusion) KIF$function)
     (forall (?g (gph.obj$object ?g))
         (and (= (KIF$source (inclusion ?g)) (diagram-fiber ?g))
              (= (KIF$target (inclusion ?g)) diagram)
              (= (inclusion ?g) (KIF$inclusion [(diagram-fiber ?g) diagram])))))

(10) (KIF$function set-fiber)
     (= (KIF$source set-fiber) gph.obj$object)
     (= (KIF$target set-fiber) KIF$function)
     (forall (?g (gph.obj$object ?g))
         (and (= (KIF$source (set-fiber ?g)) (diagram-fiber ?g))
              (= (KIF$target (set-fiber ?g)) SET.FTN$function)
              (forall (?d ((diagram-fiber ?g) ?d))
                  (= ((set-fiber ?g) ?d) (set ?d))))))

(11) (KIF$function function-fiber)
     (= (KIF$source function-fiber) gph.obj$object)
     (= (KIF$target function-fiber) KIF$function)
     (forall (?g (gph.obj$object ?g))
         (and (= (KIF$source (function-fiber ?g)) (diagram-fiber ?g))
              (= (KIF$target (function-fiber ?g)) SET.FTN$function)
              (forall (?d ((diagram-fiber ?g) ?d))
                  (= ((function-fiber ?g) ?d) (function ?d))))))
```

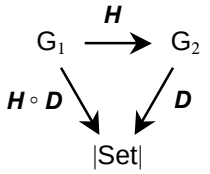


Diagram 1: Inverse Image – abstract

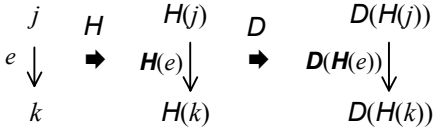


Figure 2: Inverse Image – details

Any graph morphism $H : G_1 \to G_2$ defines by composition an *inverse image* operator $H^{-1} = dgm(H) : dgm(G_2) \to dgm(G_1)$ (Figure 2) that maps any diagram $D : G_2 \to |Set|$ to the diagram $H \circ D : G_1 \to |Set|$, whose shape is (of course) the source graph G_1, whose set class function is the composition of the node function of H followed by the set class function of D, and whose function class function is the composition of the edge function of H followed by the function class function of D.

```
(12) (KIF$function inverse-image)
     (= (KIF$source inverse-image) gph.mor$morphism)
     (= (KIF$target inverse-image) SET.FTN$function)
     (forall (?h (gph.mor$morphism ?h))
         (and (= (SET.FTN$source (inverse-image ?h)) (diagram-fiber (gph.mor$target ?h)))
              (= (SET.FTN$target (inverse-image ?h)) (diagram-fiber (gph.mor$source ?h)))
              (forall (?d ((diagram-fiber (gph.mor$target ?h)) ?d))
                  (and (= ((set-fiber (gph.mor$source ?h)) ((inverse-image ?h) ?d))
                          (SET.FTN$composition
                              [(gph.mor$node ?h) ((set-fiber (gph.mor$target ?h)) ?d)]))
                       (= ((function-fiber (gph.mor$source ?h)) ((inverse-image ?h) ?d))
                          (SET.FTN$composition
                              [(gph.mor$edge ?h)
                               ((function-fiber (gph.mor$target ?h)) ?d)])))))))
```

A graph morphism and a diagram are *mixed composable* when the target of the first is equal to the graph of the second. The *mixed composition* of such a composable pair $H : G_1 \to G_2$ and $D : G_2 \to |Set|$ is defined in terms of the composition of the node/edge functions of H and the set/function functions of D.

```
(13) (SET.LIM.PBK$opspan mixed-composable-opspan)
     (= (SET.LIM.PBK$class1 mixed-composable-opspan) gph.mor$morphism)
     (= (SET.LIM.PBK$class2 mixed-composable-opspan) diagram)
     (= (SET.LIM.PBK$opvertex mixed-composable-opspan) gph.obj$object)
```

```
     (= (SET.LIM.PBK$opfirst mixed-composable-opspan) target)
     (= (SET.LIM.PBK$opsecond mixed-composable-opspan) graph)

(14) (REL$relation mixed-composable)
     (= (REL$class1 mixed-composable) gph.mor$morphism)
     (= (REL$class2 mixed-composable) diagram)
     (= (REL$extent mixed-composable) (SET.LIM.PBK$pullback mixed-composable-opspan))

(15) (SET.FTN$function mixed-composition)
     (= (SET.FTN$source mixed-composition) (SET.LIM.PBK$pullback mixed-composable-opspan))
     (= (SET.FTN$target composition) diagram)
     (forall (?h (gph.mor$morphism ?h) ?d (diagram ?d)
             (mixed-composable ?h ?d))
         (and (= (graph (mixed-composition [?h ?d])) (gph.mor$source ?h))
              (= (set (mixed-composition [?h ?d]))
                 (SET.FTN$composition [(gph.mor$node ?h) (set ?d)]))
              (= (function (mixed-composition [?h ?d]))
                 (SET.FTN$composition [(gph.mor$edge ?h) (function ?d)])))))
```

Diagram Morphisms

`set.dgm.mor`

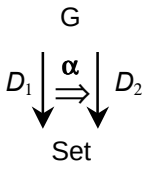


Figure 3: Diagram Morphism

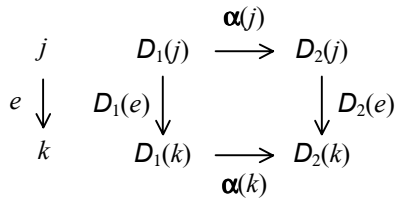


Diagram 2: Naturality

Diagrams are connected by *morphisms* or *transformations*. Just as a diagram corresponds to a functor, a diagram morphism corresponds to a natural transformation. Hence, a morphism $\alpha : D_1 \Rightarrow D_2 : \mathsf{G} \to |\mathsf{Set}|$ (Figure 3) has source and target diagrams, a shape graph, and a collection of component set functions indexed by nodes of the shape. A morphism satisfies naturality (Diagram 2).

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

(2) (KIF$function source)
    (= (KIF$source source) morphism)
    (= (KIF$target source) set.dgm$diagram)

(3) (KIF$function target)
    (= (KIF$source target) morphism)
    (= (KIF$target target) set.dgm$diagram)

(4) (KIF$function graph)
    (KIF$function shape)
    (= shape graph)
    (= (KIF$source graph) morphism)
    (= (KIF$target graph) gph.obj$object)
    (forall (?a (morphism ?a))
        (and (= (set.dgm$graph (source ?a)) (graph ?a))
             (= (set.dgm$graph (target ?a)) (graph ?a))))

(5) (KIF$function component)
    (= (KIF$source component) morphism)
    (= (KIF$target component) SET.FTN$function)
    (forall (?a (morphism ?a))
        (and (= (SET.FTN$source (component ?a)) (gph.obj$node (graph ?a)))
             (= (SET.FTN$target (component ?a)) set.mor$morphism)))
             (= (SET.FTN$composition [(component ?a) set.mor$source])
                (set.dgm$set (source ?a)))
             (= (SET.FTN$composition [(component ?a) set.mor$target])
                (set.dgm$set (target ?a)))
             (forall (?e ((gph$edge (graph ?a)) ?e))
                 (= (set.ftn$composition
                        [((component ?a) (gph$source ?e))
                         ((set.dgm$function (target ?a)) ?e)])
                    (set.ftn$composition
                        [((set.dgm$function (source ?a)) ?e)
                         ((component ?a) (gph$target ?e))])))))
```

Any diagram morphism has an associated *tuple* morphism defined by its set component.

```
(6) (KIF$function tuple)
    (= (KIF$source tuple) morphism)
    (= (KIF$target tuple) set.dgm.tpl.mor$morphism)
    (forall (?a (morphism ?a))
        (and (= (set.dgm.tpl.mor$source (tuple ?a)) (gph.dgm$tuple (source ?a)))
             (= (set.dgm.tpl.mor$target (tuple ?a)) (gph.dgm$tuple (target ?a)))
             (= (set.dgm.tpl.mor$index (tuple ?a)) (gph$node (graph ?a)))
             (= (set.dgm.tpl.mor$component (tuple ?a)) (component ?a))))
```

The *fiber* ***dgm-mor***(G) of any graph G is the class of all diagram morphisms having shape graph G.

```
(7) (KIF$function morphism-fiber)
    (= (KIF$source morphism-fiber) gph.obj$object))
    (= (KIF$target morphism-fiber) SET$class)
    (= morphism-fiber (KIF$fiber graph))

(8) (KIF$function source-fiber)
    (= (KIF$source source-fiber) gph.obj$object)
    (= (KIF$target source-fiber) SET.FTN$function)
    (forall (?g (gph.obj$object ?g))
        (and (= (SET.FTN$source (source-fiber ?g)) (morphism-fiber ?g))
             (= (SET.FTN$target (source-fiber ?g)) (set.dgm$diagram-fiber ?g))
             (forall (?a ((morphism-fiber ?g) ?a))
                 (= ((source-fiber ?g) ?a) (source ?a)))))

(9) (KIF$function target-fiber)
    (= (KIF$source target-fiber) gph.obj$object)
    (= (KIF$target target-fiber) SET.FTN$function)
    (forall (?g (gph.obj$object ?g))
        (and (= (SET.FTN$source (target-fiber ?g)) (morphism-fiber ?g))
             (= (SET.FTN$target (target-fiber ?g)) (set.dgm$diagram-fiber ?g))
             (forall (?a ((morphism-fiber ?g) ?a))
                 (= ((target-fiber ?g) ?a) (target ?a)))))

(10) (KIF$function component-fiber)
     (= (KIF$source component-fiber) gph.obj$object)
     (= (KIF$target component-fiber) KIF$function)
     (forall (?g (gph.obj$object ?g))
         (and (= (KIF$source (component-fiber ?g)) (morphism-fiber ?g))
              (= (KIF$target (component-fiber ?g)) SET.FTN$function)
              (forall (?a ((morphism-fiber ?g) ?a))
                  (and (= (SET.FTN$source ((component-fiber ?g) ?a)) (gph.obj$node (graph ?a)))
                       (= (SET.FTN$target ((component-fiber ?g) ?a)) set.mor$morphism)))
                       (= ((component-fiber ?g) ?a) (component ?a))))))
```

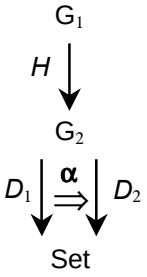


Figure 4a: Inverse Image – abstract

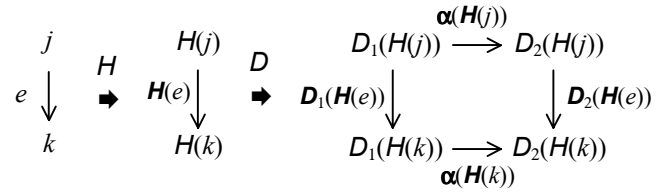


Figure 4b: Inverse Image – details

Any graph morphism $H : \mathsf{G}_1 \to \mathsf{G}_2$ defines by composition an inverse image operator H^{-1} = *dgm-mor*(H) : *dgm-mor*(G_2) → *dgm-mor*(G_1) (Figure 4) that maps any diagram morphism $\alpha : D_1 \Rightarrow D_2 : \mathsf{G}_2 \to |\mathsf{Set}|$ to the diagram morphism $\alpha : H \circ D_1 \Rightarrow H \circ D_2 : \mathsf{G}_1 \to |\mathsf{Set}|$, whose component is the composition of the node function of H followed by the component function of α.

```
(11) (KIF$function inverse-image)
     (= (KIF$source inverse-image) gph.mor$morphism)
     (= (KIF$target inverse-image) SET.FTN$function)
     (forall (?h (gph.mor$morphism ?h))
         (and (= (SET.FTN$source (inverse-image ?h)) (morphism-fiber (gph.mor$target ?h)))
              (= (SET.FTN$target (inverse-image ?h)) (morphism-fiber (gph.mor$source ?h)))
              (= (SET.FTN$composition [(inverse-image ?h) (source-fiber (gph.mor$source ?h))])
                 (SET.FTN$composition
                     [(source-fiber (gph.mor$target ?h)) (set.dgm$inverse-image ?h)]))
              (= (SET.FTN$composition [(inverse-image ?h) (target-fiber (gph.mor$source ?h))])
                 (SET.FTN$composition
                     [(target-fiber (gph.mor$target ?h)) (set.dgm$inverse-image ?h)]))
              (forall (?a ((morphism-fiber (gph.mor$target ?h)) ?a))
                  (= ((component-fiber (gph.mor$source ?h)) ((inverse-image ?h) ?a))
                     (SET.FTN$composition
```

```
                    [(gph.mor$node ?h) ((component-fiber (gph.mor$target ?h)) ?a)])))))
```

Two diagram morphisms are *composable* when the target of the first is equal to the source of the second. The *composition* of two composable diagram morphisms $\alpha : D_1 \Rightarrow D_2 : \mathsf{G} \to |\mathsf{Set}|$ and $\beta : D_2 \Rightarrow D_3 : \mathsf{G} \to |\mathsf{Set}|$ is defined in terms of the composition of their component functions.

```
(12) (SET.LIM.PBK$opspan composable-opspan)
     (= (SET.LIM.PBK$class1 composable-opspan) morphism)
     (= (SET.LIM.PBK$class2 composable-opspan) morphism)
     (= (SET.LIM.PBK$opvertex composable-opspan) set.dgm$diagram)
     (= (SET.LIM.PBK$opfirst composable-opspan) target)
     (= (SET.LIM.PBK$opsecond composable-opspan) source)

(13) (REL$relation composable)
     (= (REL$class1 composable) morphism)
     (= (REL$class2 composable) morphism)
     (= (REL$extent composable) (SET.LIM.PBK$pullback composable-opspan))

(14) (SET.FTN$function composition)
     (= (SET.FTN$source composition) (SET.LIM.PBK$pullback composable-opspan))
     (= (SET.FTN$target composition) morphism)
     (forall (?a (morphism ?a) ?b (morphism ?b)
             (composable ?a ?b))
         (and (= (source (composition [?a ?b])) (source ?a))
              (= (target (composition [?a ?b])) (target ?b))
              (forall (?n ((gph$node (graph ?a)) ?n))
                  (= ((component (composition [?a ?b])) ?n)
                     (set.mor$composition [((component ?a) ?n) ((component ?b) ?n)])))))
```

Composition satisfies the usual *associative law*.

```
     (forall (?a (morphism ?a)
              ?b (morphism ?b)
              ?c (morphism ?c)
              (composable ?a ?b) (composable ?b ?c))
         (= (composition [?a (composition [?b ?c])])
            (composition [(composition [?a ?b]) ?c])))
```

For any set diagram $D : \mathsf{G} \to |\mathsf{Set}|$, there is an *identity* diagram morphism.

```
(15) (SET.FTN$function identity)
     (= (SET.FTN$source identity) set.dgm$object)
     (= (SET.FTN$target identity) morphism)
     (forall (?d (set.dgm$diagram ?d))
         (and (= (source (identity ?d)) ?d)
              (= (target (identity ?d)) ?d)
              (= (graph (identity ?g)) (set.dgm$graph ?d))
              (forall (?n ((gph$node (set.dgm$graph ?a)) ?n))
                  (= ((component (identity ?g)) ?n)
                     (set.mor$identity ((set.dgm$set ?d) ?n))))))
```

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

```
     (forall (?a (morphism ?a))
         (and (= (composition [(identity (source ?a)) ?a]) ?a)
              (= (composition [?a (identity (target ?a))]) ?a)))
```

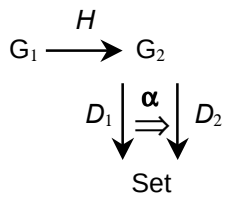


Figure 5: Diagram Morphism

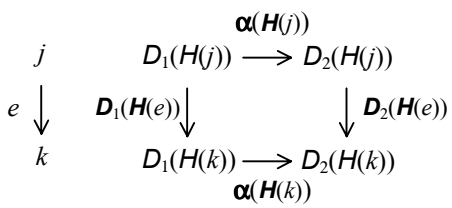


Diagram 3: Naturality

A graph morphism and a diagram morphism are *mixed composable* when the target of the first is equal to the graph of the second. The *mixed composition* of such a composable pair $H : \mathsf{G}_1 \to \mathsf{G}_2$ and

$\alpha : D_1 \Rightarrow D_2 : G_2 \to |\mathsf{Set}|$ (Figure 5) is defined in terms of the composition of the node function of H and the component function of α.

```
(16) (SET.LIM.PBK$opspan mixed-composable-opspan)
     (= (SET.LIM.PBK$class1 mixed-composable-opspan) gph.mor$morphism)
     (= (SET.LIM.PBK$class2 mixed-composable-opspan) morphism)
     (= (SET.LIM.PBK$opvertex mixed-composable-opspan) gph.obj$object)
     (= (SET.LIM.PBK$opfirst mixed-composable-opspan) gph.mor$target)
     (= (SET.LIM.PBK$opsecond mixed-composable-opspan) graph)

(17) (REL$relation mixed-composable)
     (= (REL$class1 mixed-composable) gph.mor$morphism)
     (= (REL$class2 mixed-composable) morphism)
     (= (REL$extent mixed-composable) (SET.LIM.PBK$pullback mixed-composable-opspan))

(18) (SET.FTN$function mixed-composition)
     (= (SET.FTN$source mixed-composition)
        (SET.LIM.PBK$pullback mixed-composable-opspan))
     (= (SET.FTN$target composition) morphism)
     (forall (?h (gph.mor$morphism ?h) ?a (morphism ?a)
             (mixed-composable ?h ?a))
         (and (= (source (mixed-composition [?h ?a]))
                 (set.dgm$mixed-composition [?h (source ?a)]))
              (= (target (mixed-composition [?a ?b]))
                 (set.dgm$mixed-composition [?h (target ?a)]))
              (= (component (mixed-composition [?h ?a]))
                 (SET.FTN$composition [(gph.mor$node ?h) (component ?a)])))))
```

Lax Diagram Morphisms

`set.dgm.lmor`

We define a lax version of morphism here, one that is suitable for limits. A *lax morphism* $\mathsf{F} = (H, \alpha) : (\mathsf{G}_1, D_1) \Rightarrow (\mathsf{G}_2, D_2)$ from diagram (G_1, D_1) to diagram (G_2, D_2) consists if a graph morphism $H : \mathsf{G}_1 \to \mathsf{G}_2$ and a diagram morphism $\boldsymbol{\alpha} : H \circ D_2 \Rightarrow D_1 : \mathsf{G}_1 \to |\mathsf{Set}|$ (Figure 6).

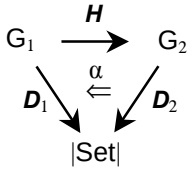


Figure 6: Lax Morphism

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

(2) (KIF$function source)
    (= (KIF$source source) lax-morphism)
    (= (KIF$target source) set.dgm$diagram)

(3) (KIF$function target)
    (= (KIF$source target) lax-morphism)
    (= (KIF$target target) set.dgm$diagram)

(4) (KIF$function graph-morphism)
    (KIF$function shape)
    (= shape graph-morphism)
    (= (KIF$source graph-morphism) lax-morphism)
    (= (KIF$target graph-morphism) gph.mor$morphism)
    (forall (?f (lax-morphism ?f))
        (and (= (gph.mor$source (graph-morphism ?f)) (set.dgm$graph (source ?f)))
             (= (gph.mor$target (graph-morphism ?f)) (set.dgm$graph (target ?f)))))

(5) (KIF$function target-diagram)
    (= (KIF$source target-diagram) lax-morphism)
    (= (KIF$target target-diagram) set.dgm$diagram)
    (forall (?f (lax-morphism ?f))
        (and (= (set.dgm$graph (target-diagram ?f)) (set.dgm$graph (source ?f)))
             (= (set.dgm$set (target-diagram ?f))
                (SET.FTN$composition
                    [(gph.mor$node (graph-morphism ?f)) (set.dgm$set (target ?f))]))
             (= (set.dgm$function (target-diagram ?f))
                (SET.FTN$composition
                    [(gph.mor$edge (graph-morphism ?f)) (set.dgm$function (target ?f))])))))

(6) (KIF$function diagram-morphism)
    (= (KIF$source diagram-morphism) lax-morphism)
    (= (KIF$target diagram-morphism) set.dgm.mor$morphism)
    (forall (?f (lax-morphism ?f))
        (and (= (set.dgm.mor$source (graph-morphism ?f)) (target-diagram ?f))
             (= (set.dgm.mor$target (graph-morphism ?f)) (source ?f))
             (= (set.dgm.mor$graph (graph-morphism ?f)) (set.dgm$graph (source ?f))))))
```

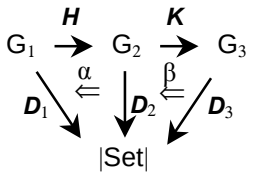


Figure 7: Lax Morphism Composition

Two lax diagram morphisms are *composable* when the target of the first is equal to the source of the second. The *composition* of two composable lax diagram morphisms $\mathsf{F} = (H, \alpha) : (\mathsf{G}_1, D_1) \Rightarrow (\mathsf{G}_2, D_2)$ and $\mathsf{G} = (K, \beta) : (\mathsf{G}_2, D_2) \Rightarrow (\mathsf{G}_3, D_3)$ (Figure 7) is defined in terms of the composition of their graph morphisms and diagram morphisms.

```
(7) (SET.LIM.PBK$opspan composable-opspan)
    (= (SET.LIM.PBK$class1 composable-opspan) lax-morphism)
    (= (SET.LIM.PBK$class2 composable-opspan) lax-morphism)
```

```
    (= (SET.LIM.PBK$opvertex composable-opspan) set.dgm$diagram)
    (= (SET.LIM.PBK$opfirst composable-opspan) target)
    (= (SET.LIM.PBK$opsecond composable-opspan) source)

 (8) (REL$relation composable)
    (= (REL$class1 composable) lax-morphism)
    (= (REL$class2 composable) lax-morphism)
    (= (REL$extent composable) (SET.LIM.PBK$pullback composable-opspan))

 (9) (SET.FTN$function composition)
    (= (SET.FTN$source composition) (SET.LIM.PBK$pullback composable-opspan))
    (= (SET.FTN$target composition) lax-morphism)
    (forall (?f (lax-morphism ?f) ?g (lax-morphism ?g)
            (composable ?f ?g))
        (and (= (source (composition [?f ?g])) (source ?f))
             (= (target (composition [?f ?g])) (target ?g))
             (= (graph-morphism (composition [?f ?g]))
                (gph.mor$composition [(graph-morphism ?f) (graph-morphism ?g)]))
             (= (diagram-morphism (composition [?f ?g]))
                (set.dgm.mor$composition
                    [(set.dgm.mor$mixed-composition
                         [(graph-morphism ?f) (diagram-morphism ?g)])
                     (diagram-morphism ?f)])))))
```

Composition satisfies the usual *associative law*.

```
    (forall (?f (lax-morphism ?f)
             ?g (lax-morphism ?g)
             ?h (lax-morphism ?h)
             (composable ?f ?g) (composable ?g ?h))
        (= (composition [?f (composition [?g ?h])])
           (composition [(composition [?f ?g]) ?h])))
```

For any set diagram $D : \mathsf{G} \to |\mathsf{Set}|$, there is an *identity* lax diagram morphism.

```
(10) (SET.FTN$function identity)
    (= (SET.FTN$source identity) set.dgm$object)
    (= (SET.FTN$target identity) lax-morphism)
    (forall (?d (set.dgm$diagram ?d))
        (and (= (source (identity ?d)) ?d)
             (= (target (identity ?d)) ?d)
             (= (graph-morphism (identity ?d)) (gph.mor$identity (set.dgm$graph ?d)))
             (= (diagram-morphism (identity ?d)) (set.dgm.mor$identity ?d))))
```

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

```
    (forall (?f (lax-morphism ?f))
        (and (= (composition [(identity (source ?f)) ?f]) ?f)
             (= (composition [?f (identity (target ?f))]) ?f)))
```

Colax Diagram Morphisms

`set.dgm.clmor`

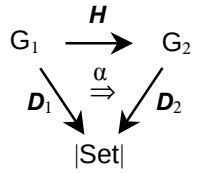


Figure 8: Colax Morphism

We define a lax version of morphism here, one that is suitable for colimits. A *colax morphism* $\mathsf{F} = (H, \alpha) : (\mathsf{G}_1, D_1) \Rightarrow (\mathsf{G}_2, D_2)$ from diagram (G_1, D_1) to diagram (G_2, D_2) consists of a graph morphism $H : \mathsf{G}_1 \to \mathsf{G}_2$ and a diagram morphism $\boldsymbol{\alpha} : D_1 \Rightarrow H \circ D_2 : \mathsf{G} \to |\mathsf{Set}|$ (Figure 8).

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

(2) (KIF$function source)
    (= (KIF$source source) colax-morphism)
    (= (KIF$target source) set.dgm$diagram)

(3) (KIF$function target)
    (= (KIF$source target) colax-morphism)
    (= (KIF$target target) set.dgm$diagram)

(4) (KIF$function graph-morphism)
    (KIF$function shape)
    (= shape graph-morphism)
    (= (KIF$source graph-morphism) colax-morphism)
    (= (KIF$target graph-morphism) gph.mor$morphism)
    (forall (?f (colax-morphism ?f))
        (and (= (gph.mor$source (graph-morphism ?f)) (set.dgm$graph (source ?f)))
             (= (gph.mor$target (graph-morphism ?f)) (set.dgm$graph (target ?f)))))

(5) (KIF$function target-diagram)
    (= (KIF$source target-diagram) colax-morphism)
    (= (KIF$target target-diagram) set.dgm$diagram)
    (forall (?f (colax-morphism ?f))
        (and (= (set.dgm$graph (target-diagram ?f)) (set.dgm$graph (source ?f)))
             (= (set.dgm$set (target-diagram ?f))
                (SET.FTN$composition
                    [(gph.mor$node (graph-morphism ?f)) (set.dgm$set (target ?f))]))
             (= (set.dgm$function (target-diagram ?f))
                (SET.FTN$composition
                    [(gph.mor$edge (graph-morphism ?f)) (set.dgm$function (target ?f))])))))

(6) (KIF$function diagram-morphism)
    (= (KIF$source diagram-morphism) colax-morphism)
    (= (KIF$target diagram-morphism) set.dgm.mor$morphism)
    (forall (?f (colax-morphism ?f))
        (and (= (set.dgm.mor$source (graph-morphism ?f)) (source ?f))
             (= (set.dgm.mor$target (graph-morphism ?f)) (target-diagram ?f))
             (= (set.dgm.mor$graph (graph-morphism ?f)) (set.dgm$graph (source ?f))))))
```

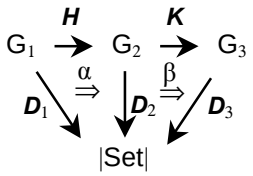


Figure 9: Colax Morphism Composition

Two colax diagram morphisms are *composable* when the target of the first is equal to the source of the second. The *composition* of two composable colax diagram morphisms $\mathsf{F} = (H, \alpha) : (\mathsf{G}_1, D_1) \Rightarrow (\mathsf{G}_2, D_2)$ and $\mathsf{G} = (K, \beta) : (\mathsf{G}_2, D_2) \Rightarrow (\mathsf{G}_3, D_3)$ (Figure 9) is defined in terms of the composition of their graph morphisms and diagram morphisms.

```
(7) (SET.LIM.PBK$opspan composable-opspan)
    (= (SET.LIM.PBK$class1 composable-opspan) colax-morphism)
    (= (SET.LIM.PBK$class2 composable-opspan) colax-morphism)
```

```
    (= (SET.LIM.PBK$opvertex composable-opspan) set.dgm$diagram)
    (= (SET.LIM.PBK$opfirst composable-opspan) target)
    (= (SET.LIM.PBK$opsecond composable-opspan) source)

(8) (REL$relation composable)
    (= (REL$class1 composable) colax-morphism)
    (= (REL$class2 composable) colax-morphism)
    (= (REL$extent composable) (SET.LIM.PBK$pullback composable-opspan))

(9) (SET.FTN$function composition)
    (= (SET.FTN$source composition) (SET.LIM.PBK$pullback composable-opspan))
    (= (SET.FTN$target composition) colax-morphism)
    (forall (?f (colax-morphism ?f) ?g (colax-morphism ?g)
            (composable ?f ?g))
        (and (= (source (composition [?f ?g])) (source ?f))
             (= (target (composition [?f ?g])) (target ?g))
             (= (graph-morphism (composition [?f ?g]))
                (gph.mor$composition [(graph-morphism ?f) (graph-morphism ?g)]))
             (= (diagram-morphism (composition [?f ?g]))
                (set.dgm.mor$composition
                    [(diagram-morphism ?f)
                     (set.dgm.mor$mixed-composition
                         [(graph-morphism ?f) (diagram-morphism ?g)])])))))
```

Composition satisfies the usual *associative law*.

```
    (forall (?f (colax-morphism ?f)
             ?g (colax-morphism ?g)
             ?h (colax-morphism ?h)
             (composable ?f ?g) (composable ?g ?h))
        (= (composition [?f (composition [?g ?h])])
           (composition [(composition [?f ?g]) ?h])))
```

For any set diagram $D : \mathsf{G} \to |\mathsf{Set}|$, there is an *identity* colax diagram morphism.

```
(10) (SET.FTN$function identity)
    (= (SET.FTN$source identity) set.dgm$object)
    (= (SET.FTN$target identity) colax-morphism)
    (forall (?d (set.dgm$diagram ?d))
        (and (= (source (identity ?d)) ?d)
             (= (target (identity ?d)) ?d)
             (= (graph-morphism (identity ?d)) (gph.mor$identity (set.dgm$graph ?d)))
             (= (diagram-morphism (identity ?d)) (set.dgm.mor$identity ?d))))
```

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

```
    (forall (?f (colax-morphism ?f))
        (and (= (composition [(identity (source ?f)) ?f]) ?f)
             (= (composition [?f (identity (target ?f))]) ?f)))
```

Tuples

set.dgm.tpl

A *tuple* of sets $\boldsymbol{A} = \{\boldsymbol{A}(n) \mid n \in J\}$ (Figure 10) is a function from a (small) *index* set J to set = set*obj*(Set) the object class of sets. Therefore, any tuple of sets is a class function $J \to \mathsf{set}$. We regard a tuple of sets to be a discrete diagram, a diagram with a discrete shape graph J. As such, the edge set of the shape graph is empty, and we only name the node set, not the graph itself. This node set is called the *index* set. In addition, the node-set class function is called the *set* function. We ignore the edge-function class function (of the diagram), since it is empty. Hence, axiomatically, a *tuple* of sets $\boldsymbol{A} = \langle \mathit{ind}(\boldsymbol{A}), \mathit{set}(\boldsymbol{A}) \rangle$ consists of an *index* set $J = \mathit{ind}(\boldsymbol{A})$ and a *set* class function $\mathit{set}(\boldsymbol{A}) : J \to \mathit{obj}(\mathsf{Set}) = \mathsf{set}$. Since each tuple is a class function, the collection of all the tuple sets forms a collection, which is a subcollection of the collection of class functions.

$$J \xrightarrow{\boldsymbol{A}} \mathsf{set}$$

Figure 10: Tuple

```
(1) (KIF$collection tuple)
    (KIF$collection object)
    (= object tuple)
    (KIF$subcollection tuple SET.FTN$function)

(2) (KIF$function index)
    (= (KIF$source index) tuple)
    (= (KIF$target index) set.obj$object)

(3) (KIF$function set)
    (= (KIF$source set) tuple)
    (= (KIF$target set) SET.FTN$function)
    (forall (?a (tuple ?a))
        (and (= (SET.FTN$source (set ?a)) (index ?a))
             (= (SET.FTN$target (set ?a)) set.obj$object)))
```

For any tuple of sets $\boldsymbol{A} : J \to \mathsf{set}$ there is a discrete *diagram* whose shape graph is the discrete graph of the index set $J = \mathit{ind}(\boldsymbol{A})$, whose object function (set) is $\mathit{set}(\boldsymbol{A})$ and whose morphism function (function) is empty.

```
(4) (KIF$function diagram)
    (= (KIF$source diagram) tuple)
    (= (KIF$target diagram) set.dgm$diagram)
    (forall (?a (tuple ?a))
        (and (= (set.dgm$graph (diagram ?a)) (set.obj$graph (index ?a)))
             (= (set.dgm$set (diagram ?a)) (set ?a))))
```

The diagram function is injective – the tuple of the diagram of a tuple is the original. As axiomatized in the diagram namespace, the image of the diagram function is the collection of *discrete diagrams*. Discrete diagrams (diagrams of discrete shape) are in bijective correspondence with tuples.

```
(5) (forall (?a (tuple ?a))
         (= (set.dgm$tuple (diagram ?a)) ?a))
```

The *fiber* $\mathit{tpl}(J) \subseteq \mathit{tpl}$ for any set J is the class of all tuples having index set J.

```
(6) (KIF$function tuple-fiber)
    (= (KIF$source tuple-fiber) set.obj$object))
    (= (KIF$target tuple-fiber) SET$class)
    (= tuple-fiber (KIF$fiber index))

(7) (KIF$function inclusion)
    (= (KIF$source inclusion) set.obj$object))
    (= (KIF$target inclusion) KIF$function)
    (forall (?j (set.obj$object ?j))
        (and (= (KIF$source (inclusion ?j)) (tuple-fiber ?j))
             (= (KIF$target (inclusion ?j)) tuple)
             (= (inclusion ?j) (KIF$inclusion [(tuple-fiber ?j) tuple])))))

(8) (KIF$function set-fiber)
    (= (KIF$source set-fiber) set.obj$object)
    (= (KIF$target set-fiber) KIF$function)
    (forall (?j (set.obj$object ?j))
```

```
(and (= (KIF$source (set-fiber ?j)) (tuple-fiber ?j))
     (= (KIF$target (set-fiber ?j)) SET.FTN$function)
     (forall (?a ((tuple-fiber ?j) ?a))
         (= ((set-fiber ?j) ?a) (set ?a)))))
```

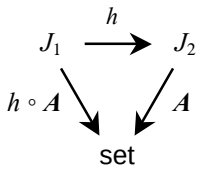


Diagram 4: Inverse Image – abstract

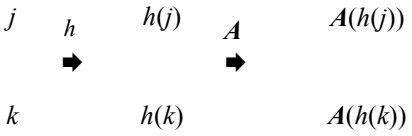


Figure 11: Inverse Image – details

Any set function $h : J_1 \to J_2$ defines by composition an *inverse image* operator $h^{-1} = \mathit{tpl}(h) : \mathit{tpl}(J_2) \to \mathit{tpl}(J_1)$ (Figure 11) that maps any tuple $\boldsymbol{A} : J_2 \to \mathsf{set}$ to the tuple $h \circ \boldsymbol{A} : J_1 \to \mathsf{set}$, whose index is (of course) the source set J_1, whose set class function is the composition of h with the set class function of $\boldsymbol{A}$.

```
(9) (KIF$function inverse-image)
    (= (KIF$source inverse-image) set.mor$morphism)
    (= (KIF$target inverse-image) SET.FTN$function)
    (forall (?h (set.mor$morphism ?h))
        (and (= (SET.FTN$source (inverse-image ?h)) (tuple-fiber (set.mor$target ?h)))
             (= (SET.FTN$target (inverse-image ?h)) (tuple-fiber (set.mor$source ?h)))
             (forall (?a ((tuple-fiber (set.mor$target ?h)) ?a))
                 (= ((set-fiber (set.mor$source ?h)) ((inverse-image ?h) ?a))
                    (SET.FTN$composition [?h ((set-fiber (set.mor$target ?h)) ?a)]))))))
```

A set function and a tuple are *mixed composable* when the target of the first is equal to the index of the second. The *mixed composition* of such a composable pair $h : J_1 \to J_2$ and $\boldsymbol{A} : J_2 \to \mathsf{set}$ is defined in terms of the composition of h with the set function of $\boldsymbol{A}$.

```
(10) (SET.LIM.PBK$opspan mixed-composable-opspan)
     (= (SET.LIM.PBK$class1 mixed-composable-opspan) set.mor$morphism)
     (= (SET.LIM.PBK$class2 mixed-composable-opspan) tuple)
     (= (SET.LIM.PBK$opvertex mixed-composable-opspan) set.obj$object)
     (= (SET.LIM.PBK$opfirst mixed-composable-opspan) set.mor$target)
     (= (SET.LIM.PBK$opsecond mixed-composable-opspan) index)

(11) (REL$relation mixed-composable)
     (= (REL$class1 mixed-composable) set.mor$morphism)
     (= (REL$class2 mixed-composable) tuple)
     (= (REL$extent mixed-composable) (SET.LIM.PBK$pullback mixed-composable-opspan))

(12) (SET.FTN$function mixed-composition)
     (= (SET.FTN$source mixed-composition) (SET.LIM.PBK$pullback mixed-composable-opspan))
     (= (SET.FTN$target composition) tuple)
     (forall (?h (set.mor$morphism ?h) ?a (tuple ?a)
             (mixed-composable ?h ?a))
         (and (= (index (mixed-composition [?h ?a])) (set.mor$source ?h))
              (= (set (mixed-composition [?h ?a]))
                 (SET.FTN$composition [?h (set ?a)])))))
```

Tuple Morphisms

`set.dgm.tpl.mor`

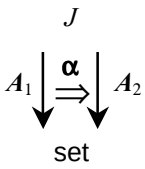


Figure 12: Tuple Morphism

$$\begin{array}{lcc} & \overset{\alpha(j)}{\longrightarrow} & \\ j & A_1(j) \longrightarrow & A_2(j) \\ k & A_1(k) \longrightarrow & A_2(k) \\ & \underset{\alpha(k)}{\longrightarrow} & \end{array}$$

Diagram 5: Detail

Tuples are connected by *morphisms*. Just as a tuple corresponds to a discrete diagram, a tuple morphism corresponds to a discrete diagram morphism – since edge functions do not exist, naturality is vacuous. Hence, a morphism $\alpha : A_1 \Rightarrow A_2 : J \to \mathsf{set}$ (Figure 12) has source and target tuples, an index set, and a collection of component set functions (Diagram 5) indexed by elements of the index.

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

(2) (KIF$function source)
    (= (KIF$source source) morphism)
    (= (KIF$target source) set.dgm.tpl$tuple)

(3) (KIF$function target)
    (= (KIF$source target) morphism)
    (= (KIF$target target) set.dgm.tpl$tuple)

(4) (KIF$function index)
    (= (KIF$source index) morphism)
    (= (KIF$target index) set.obj$object)
    (forall (?a (morphism ?a))
        (and (= (set.dgm.tpl$index (source ?a)) (index ?a))
             (= (set.dgm.tpl$index (target ?a)) (index ?a))))

(5) (KIF$function component)
    (= (KIF$source component) morphism)
    (= (KIF$target component) SET.FTN$function)
    (forall (?a (morphism ?a))
        (and (= (SET.FTN$source (component ?a)) (index ?a))
             (= (SET.FTN$target (component ?a)) set.mor$morphism)))
             (= (SET.FTN$composition [(component ?a) set.mor$source])
                (set.dgm.tpl$set (source ?a)))
             (= (SET.FTN$composition [(component ?a) set.mor$target])
                (set.dgm.tpl$set (target ?a)))))
```

For any tuple morphism $\alpha : A_1 \Rightarrow A_2 : J \to \mathsf{set}$, there is a morphism of discrete *diagrams* whose source is the discrete diagram of the source, whose target is the discrete diagram of the target, whose shape graph is the discrete graph of the index set J, and whose component function is the component function of α.

```
(6) (KIF$function diagram)
    (= (KIF$source diagram) morphism)
    (= (KIF$target diagram) set.dgm.mor$morphism)
    (forall (?a (morphism ?a))
        (and (= (set.dgm.mor$source (diagram ?a)) (set.dgm.tpl$diagram (source ?a)))
             (= (set.dgm.mor$target (diagram ?a)) (set.dgm.tpl$diagram (target ?a)))
             (= (set.dgm.mor$graph (diagram ?a)) (set.obj$graph (index ?a)))
             (= (set.dgm.mor$component (diagram ?a)) (component ?a))))
```

The diagram function is injective – the tuple morphism of the diagram morphism of a tuple morphism is the original. Discrete diagram morphisms (morphism between diagrams of discrete shape) are in bijective correspondence with tuple morphisms.

```
(7) (forall (?a (morphism ?a))
        (= (set.dgm.mor$tuple (diagram ?a)) ?a))
```

Two tuple morphisms are *composable* when the target of the first is equal to the source of the second. The *composition* of two composable tuple morphisms $\alpha : A_1 \Rightarrow A_2 : J \to \mathsf{set}$ and $\beta : A_2 \Rightarrow A_3 : J \to \mathsf{set}$ is defined in terms of the composition of their component functions.

```
 (8) (SET.LIM.PBK$opspan composable-opspan)
     (= (SET.LIM.PBK$class1 composable-opspan) morphism)
     (= (SET.LIM.PBK$class2 composable-opspan) morphism)
     (= (SET.LIM.PBK$opvertex composable-opspan) set.dgm.tpl$tuple)
     (= (SET.LIM.PBK$opfirst composable-opspan) target)
     (= (SET.LIM.PBK$opsecond composable-opspan) source)

 (9) (REL$relation composable)
     (= (REL$class1 composable) morphism)
     (= (REL$class2 composable) morphism)
     (= (REL$extent composable) (SET.LIM.PBK$pullback composable-opspan))

(10) (SET.FTN$function composition)
     (= (SET.FTN$source composition) (SET.LIM.PBK$pullback composable-opspan))
     (= (SET.FTN$target composition) morphism)
     (forall (?a (morphism ?a) ?b (morphism ?b)
               (composable ?a ?b))
         (and (= (source (composition [?a ?b])) (source ?a))
              (= (target (composition [?a ?b])) (target ?b))
              (forall (?n ((index ?a) ?n))
                  (= ((component (composition [?a ?b])) ?n)
                     (set.mor$composition [((component ?a) ?n) ((component ?b) ?n)])))))
```

Composition satisfies the usual *associative law*.

```
     (forall (?a (morphism ?a)
              ?b (morphism ?b)
              ?c (morphism ?c)
              (composable ?a ?b) (composable ?b ?c))
         (= (composition [?a (composition [?b ?c])])
            (composition [(composition [?a ?b]) ?c])))
```

For any set tuple $A : J \to \mathsf{set}$, there is an *identity* tuple morphism.

```
(11) (SET.FTN$function identity)
     (= (SET.FTN$source identity) set.dgm.tpl$tuple)
     (= (SET.FTN$target identity) morphism)
     (forall (?a (set.dgm.tpl$tuple ?a))
         (and (= (source (identity ?a)) ?a)
              (= (target (identity ?a)) ?a)
              (= (index (identity ?a)) (set.dgm.tpl$index ?a))
              (forall (?n ((set.dgm.tpl$index ?a) ?n))
                  (= ((component (identity ?a)) ?n)
                     (set.mor$identity ((set.dgm.tpl$set ?a) ?n))))))
```

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

```
     (forall (?a (morphism ?a))
         (and (= (composition [(identity (source ?a)) ?a]) ?a)
              (= (composition [?a (identity (target ?a))]) ?a)))
```

A set function and a tuple morphism are *mixed composable* when the target of the first is equal to the index of the second. The *mixed composition* of such a composable pair $h : J_1 \to J_2$ and $\alpha : A_1 \Rightarrow A_2 : J_2 \to \mathsf{set}$ (Figure 13) is defined in terms of the composition of h with the component function of α.

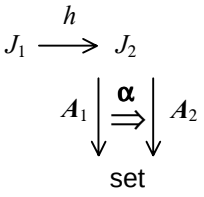


Figure 13: Tuple Morphism

```
(12) (SET.LIM.PBK$opspan mixed-composable-opspan)
     (= (SET.LIM.PBK$class1 mixed-composable-opspan) set.mor$morphism)
     (= (SET.LIM.PBK$class2 mixed-composable-opspan) morphism)
     (= (SET.LIM.PBK$opvertex mixed-composable-opspan) set.obj$object)
     (= (SET.LIM.PBK$opfirst mixed-composable-opspan) set.mor$target)
     (= (SET.LIM.PBK$opsecond mixed-composable-opspan) index)
```

```
(13) (REL$relation mixed-composable)
     (= (REL$class1 mixed-composable) set.mor$morphism)
     (= (REL$class2 mixed-composable) morphism)
     (= (REL$extent mixed-composable) (SET.LIM.PBK$pullback mixed-composable-opspan))

(14) (SET.FTN$function mixed-composition)
     (= (SET.FTN$source mixed-composition)
        (SET.LIM.PBK$pullback mixed-composable-opspan))
     (= (SET.FTN$target composition) morphism)
     (forall (?h (set.mor$morphism ?h) ?a (morphism ?a)
             (mixed-composable ?h ?a))
         (and (= (source (mixed-composition [?h ?a]))
                 (set.dgm.tpl$mixed-composition [?h (source ?a)]))
              (= (target (mixed-composition [?a ?b]))
                 (set.dgm.tpl$mixed-composition [?h (target ?a)]))
              (= (component (mixed-composition [?h ?a]))
                 (SET.FTN$composition [?h (component ?a)])))))
```

Lax Tuple Morphisms

`set.dgm.tpl.lmor`

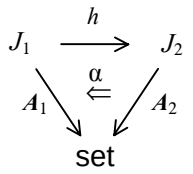


Figure 14: Lax Morphism

We define a lax version of morphism here, one that is suitable for limits. A *lax morphism* $\mathsf{F} = (h, \alpha) : (J_1, \boldsymbol{A}_1) \Rightarrow (J_2, \boldsymbol{A}_2)$ from tuple $(J_1, \boldsymbol{A}_1)$ to tuple $(J_2, \boldsymbol{A}_2)$ consists if a set function $h : J_1 \to J_2$ and a tuple morphism $\boldsymbol{\alpha} : h \circ \boldsymbol{A}_2 \Rightarrow \boldsymbol{A}_1 : J_1 \to \mathsf{set}$ (Figure 14).

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

(2) (KIF$function source)
    (= (KIF$source source) lax-morphism)
    (= (KIF$target source) set.dgm.tpl$tuple)

(3) (KIF$function target)
    (= (KIF$source target) lax-morphism)
    (= (KIF$target target) set.dgm.tpl$tuple)

(4) (KIF$function set-function)
    (KIF$function index)
    (= index set-function)
    (= (KIF$source set-function) lax-morphism)
    (= (KIF$target set-function) set.mor$morphism)
    (forall (?f (lax-morphism ?f))
        (and (= (set.mor$source (set-function ?f)) (set.dgm.tpl$index (source ?f)))
             (= (set.mor$target (set-function ?f)) (set.dgm.tpl$index (target ?f))))))

(5) (KIF$function target-tuple)
    (= (KIF$source target-tuple) lax-morphism)
    (= (KIF$target target-tuple) set.dgm.tpl$tuple)
    (forall (?f (lax-morphism ?f))
        (and (= (set.dgm.tpl$index (target-tuple ?f)) (set.dgm.tpl$index (source ?f)))
             (= (set.dgm.tpl$set (target-tuple ?f))
                (SET.FTN$composition
                    [(set-function ?f) (set.dgm.tpl$set (target ?f))])))))

(6) (KIF$function tuple-morphism)
    (= (KIF$source tuple-morphism) lax-morphism)
    (= (KIF$target tuple-morphism) set.dgm.tpl.mor$morphism)
    (forall (?f (lax-morphism ?f))
        (and (= (set.dgm.tpl.mor$source (tuple-morphism ?f)) (target-diagram ?f))
             (= (set.dgm.tpl.mor$target (tuple-morphism ?f)) (source ?f))
             (= (set.dgm.tpl.mor$index (tuple-morphism ?f)) (set.dgm.tpl$index (source ?f))))))
```

Two lax tuple morphisms are *composable* when the target of the first is equal to the source of the second.

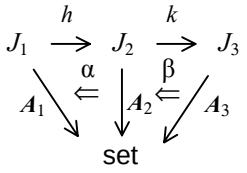


Figure 15: Lax Morphism Composition

The *composition* of two composable lax tuple morphisms $\mathsf{F} = (h, \alpha) : (J_1, \boldsymbol{A}_1) \Rightarrow (J_2, \boldsymbol{A}_2)$ and $\mathsf{G} = (k, \beta) : (J_2, \boldsymbol{A}_2) \Rightarrow (J_3, \boldsymbol{A}_3)$ (Figure 15) is defined in terms of the composition of their set functions and tuple morphisms.

```
(7) (SET.LIM.PBK$opspan composable-opspan)
    (= (SET.LIM.PBK$class1 composable-opspan) lax-morphism)
    (= (SET.LIM.PBK$class2 composable-opspan) lax-morphism)
    (= (SET.LIM.PBK$opvertex composable-opspan) set.dgm.tpl$tuple)
    (= (SET.LIM.PBK$opfirst composable-opspan) target)
    (= (SET.LIM.PBK$opsecond composable-opspan) source)

(8) (REL$relation composable)
    (= (REL$class1 composable) lax-morphism)
    (= (REL$class2 composable) lax-morphism)
    (= (REL$extent composable) (SET.LIM.PBK$pullback composable-opspan))

(9) (SET.FTN$function composition)
    (= (SET.FTN$source composition) (SET.LIM.PBK$pullback composable-opspan))
    (= (SET.FTN$target composition) lax-morphism)
    (forall (?f (lax-morphism ?f) ?g (lax-morphism ?g)
            (composable ?f ?g))
        (and (= (source (composition [?f ?g])) (source ?f))
             (= (target (composition [?f ?g])) (target ?g))
             (= (set-function (composition [?f ?g]))
                (set.mor$composition [(set-function ?f) (set-function ?g)]))
             (= (tuple-morphism (composition [?f ?g]))
                (set.dgm.mor$composition
                    [(set.dgm.tpl.mor$mixed-composition
                         [(set-function ?f) (tuple-morphism ?g)])
                     (tuple-morphism ?f)])))))
```

Composition satisfies the usual *associative law*.

```
    (forall (?f (lax-morphism ?f)
             ?g (lax-morphism ?g)
             ?h (lax-morphism ?h)
             (composable ?f ?g) (composable ?g ?h))
        (= (composition [?f (composition [?g ?h])])
           (composition [(composition [?f ?g]) ?h])))
```

For any set tuple $\boldsymbol{A} : J \to \mathsf{set}$, there is an *identity* lax tuple morphism.

```
(10) (SET.FTN$function identity)
     (= (SET.FTN$source identity) set.dgm.tpl$tuple)
     (= (SET.FTN$target identity) lax-morphism)
     (forall (?a (set.dgm.tpl$tuple ?a))
         (and (= (source (identity ?a)) ?a)
              (= (target (identity ?a)) ?a)
              (= (set-function (identity ?a)) (set.mor$identity (set.dgm.tpl$index ?a)))
              (= (tuple-morphism (identity ?a)) (set.dgm.tpl.mor$identity ?a))))
```

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

```
    (forall (?f (lax-morphism ?f))
        (and (= (composition [(identity (source ?f)) ?f]) ?f)
             (= (composition [?f (identity (target ?f))]) ?f)))
```

Colax Tuple Morphisms

`set.dgm.tpl.clmor`

We define a lax version of morphism here, one that is suitable for colimits. A *colax morphism* $\mathsf{F} = (h, \alpha) : (J_1, \boldsymbol{A}_1) \Rightarrow (J_2, \boldsymbol{A}_2)$ from tuple $(J_1, \boldsymbol{A}_1)$ to tuple $(J_2, \boldsymbol{A}_2)$ consists if a set function $h : J_1 \to J_2$ and a tuple morphism $\boldsymbol{\alpha} : \boldsymbol{A}_1 \Rightarrow h \circ \boldsymbol{A}_2 : J_1 \to \mathsf{set}$ (Figure 16).

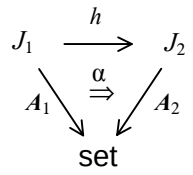


Figure 16: Lax Morphism

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

(2) (KIF$function source)
    (= (KIF$source source) colax-morphism)
    (= (KIF$target source) set.dgm.tpl$tuple)

(3) (KIF$function target)
    (= (KIF$source target) colax-morphism)
    (= (KIF$target target) set.dgm.tpl$tuple)

(4) (KIF$function set-function)
    (KIF$function index)
    (= index set-function)
    (= (KIF$source set-function) colax-morphism)
    (= (KIF$target set-function) set.mor$morphism)
    (forall (?f (colax-morphism ?f))
        (and (= (set.mor$source (set-function ?f)) (set.dgm.tpl$index (source ?f)))
             (= (set.mor$target (set-function ?f)) (set.dgm.tpl$index (target ?f)))))

(5) (KIF$function target-tuple)
    (= (KIF$source target-tuple) colax-morphism)
    (= (KIF$target target-tuple) set.dgm.tpl$tuple)
    (forall (?f (colax-morphism ?f))
        (and (= (set.dgm.tpl$index (target-tuple ?f)) (set.dgm.tpl$index (source ?f)))
             (= (set.dgm.tpl$set (target-tuple ?f))
                (SET.FTN$composition
                    [(set-function ?f) (set.dgm.tpl$set (target ?f))])))))

(6) (KIF$function tuple-morphism)
    (= (KIF$source tuple-morphism) colax-morphism)
    (= (KIF$target tuple-morphism) set.dgm.tpl.mor$morphism)
    (forall (?f (colax-morphism ?f))
        (and (= (set.dgm.tpl.mor$source (tuple-morphism ?f)) (source ?f))
             (= (set.dgm.tpl.mor$target (tuple-morphism ?f)) (target-diagram ?f))
             (= (set.dgm.tpl.mor$index (tuple-morphism ?f)) (set.dgm.tpl$index (source ?f))))))
```

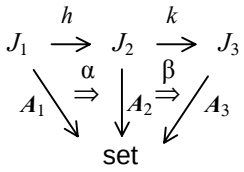


Figure 17: Colax Morphism Composition

Two colax tuple morphisms are *composable* when the target of the first is equal to the source of the second. The *composition* of two composable lax tuple morphisms $\mathsf{F} = (h, \alpha) : (J_1, \boldsymbol{A}_1) \Rightarrow (J_2, \boldsymbol{A}_2)$ and $\mathsf{G} = (k, \beta) : (J_2, \boldsymbol{A}_2) \Rightarrow (J_3, \boldsymbol{A}_3)$ (Figure 17) is defined in terms of the composition of their set functions and tuple morphisms.

```
(7) (SET.LIM.PBK$opspan composable-opspan)
    (= (SET.LIM.PBK$class1 composable-opspan) colax-morphism)
    (= (SET.LIM.PBK$class2 composable-opspan) colax-morphism)
    (= (SET.LIM.PBK$opvertex composable-opspan) set.dgm.tpl$tuple)
    (= (SET.LIM.PBK$opfirst composable-opspan) target)
    (= (SET.LIM.PBK$opsecond composable-opspan) source)

(8) (REL$relation composable)
```

```
    (= (REL$class1 composable) colax-morphism)
    (= (REL$class2 composable) colax-morphism)
    (= (REL$extent composable) (SET.LIM.PBK$pullback composable-opspan))

(9) (SET.FTN$function composition)
    (= (SET.FTN$source composition) (SET.LIM.PBK$pullback composable-opspan))
    (= (SET.FTN$target composition) colax-morphism)
    (forall (?f (colax-morphism ?f) ?g (colax-morphism ?g)
            (composable ?f ?g))
        (and (= (source (composition [?f ?g])) (source ?f))
             (= (target (composition [?f ?g])) (target ?g))
             (= (set-function (composition [?f ?g]))
                (set.mor$composition [(set-function ?f) (set-function ?g)]))
             (= (tuple-morphism (composition [?f ?g]))
                (set.dgm.mor$composition
                    [(tuple-morphism ?f)
                     (set.dgm.tpl.mor$mixed-composition
                         [(set-function ?f) (tuple-morphism ?g)])])))))
```

Composition satisfies the usual *associative law*.

```
    (forall (?f (colax-morphism ?f)
             ?g (colax-morphism ?g)
             ?h (colax-morphism ?h)
             (composable ?f ?g) (composable ?g ?h))
        (= (composition [?f (composition [?g ?h])])
           (composition [(composition [?f ?g]) ?h])))
```

For any set tuple $\boldsymbol{A} : J \to \mathsf{set}$, there is an *identity* colax tuple morphism.

```
(10) (SET.FTN$function identity)
    (= (SET.FTN$source identity) set.dgm.tpl$tuple)
    (= (SET.FTN$target identity) colax-morphism)
    (forall (?a (set.dgm.tpl$tuple ?a))
        (and (= (source (identity ?a)) ?a)
             (= (target (identity ?a)) ?a)
             (= (set-function (identity ?a)) (set.mor$identity (set.dgm.tpl$index ?a)))
             (= (tuple-morphism (identity ?a)) (set.dgm.tpl.mor$identity ?a))))
```

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

```
    (forall (?f (colax-morphism ?f))
        (and (= (composition [(identity (source ?f)) ?f]) ?f)
             (= (composition [?f (identity (target ?f))]) ?f)))
```

Arities

`set.dgm.art`

Discrete based diagrams are in bijective correspondence with tuples of sets. These are axiomatized in the tuple sub-namespace. However, many of the discrete diagrams or tuples used in the IFF Ontology (meta) Ontology are based. Based tuples of sets are called tuple subsets. Another name for tuple subsets is arities, so-named because many examples come from relation-like arity functions. Arity is the name that we use for tuple subsets in the IFF. An arity is the appropriate diagram for a based coproduct, a coproduct of subsets of a particular set. That is, an arity is an indexed collection of subsets of a particular base set. We use either the generic term '`object`' or the specific term '`arity`' to denote the *arity* class. An arity is the special case of a general diagram of discrete shape that is based. An *arity* or *tuple of subsets* $\boldsymbol{A} = \langle J, B, A \rangle = \langle \mathit{ind}(\boldsymbol{A}), \mathit{base}(\boldsymbol{A}), \mathit{set}(\boldsymbol{A}) \rangle$ is a set function $A : J \to \wp B$; in other notation, $\boldsymbol{A} = \{\boldsymbol{A}(n) \mid n \in J, \boldsymbol{A}(n) \subseteq B\}$. It consists of an *index* set $J = \mathit{ind}(\boldsymbol{A})$, a *base* set $B = \mathit{base}(\boldsymbol{A})$ and *set* function $\boldsymbol{A} : J \to \wp B$. Since an arity is a set function, the collection of all the arities forms a class, not a generic collection. Arities are determined by their arity-base-set triples. This fact is represented by an injection assertion for the function $\mathit{set} : \mathsf{arity} \to \mathsf{ftn}$.

```
(1) (SET$class arity)
    (SET$class object)
    (= object arity)

(2) (SET.FTN$function index)
    (= (SET.FTN$source index) arity)
    (= (SET.FTN$target index) set.obj$object)

(3) (SET.FTN$function base)
    (= (SET.FTN$source base) arity)
    (= (SET.FTN$target base) set.obj$object)

(4) (SET.FTN$function set)
    (= (SET.FTN$source set) arity)
    (= (SET.FTN$target set) set.mor$morphism)
    (= (SET.FTN$composition [set set.mor$source]) index)
    (= (SET.FTN$composition [set set.mor$target])
       (SET.FTN$composition [base set$power]))

(5) (SET.FTN$injection set)
```

The *fiber* $\mathit{art}(J) \subseteq \mathit{art}$ for any set J is the class of all arities having index set J.

```
(6) (SET.FTN$function arity-fiber)
    (= (SET.FTN$source arity-fiber) set.obj$object))
    (= (SET.FTN$target arity-fiber) SET$class)
    (= arity-fiber (SET.FTN$fiber index))

(7) (SET.FTN$function inclusion)
    (= (SET.FTN$source inclusion) set.obj$object))
    (= (SET.FTN$target inclusion) SET.FTN$function)
    (forall (?j (set.obj$object ?j))
        (and (= (SET.FTN$source (inclusion ?j)) (arity-fiber ?j))
             (= (SET.FTN$target (inclusion ?j)) arity)
             (= (inclusion ?j) (SET.FTN$inclusion [(arity-fiber ?j) arity]))))

(8) (SET.FTN$function base-fiber)
    (= (SET.FTN$source base-fiber) set.obj$object)
    (= (SET.FTN$target base-fiber) SET.FTN$function)
    (forall (?j (set.obj$object ?j))
        (and (= (SET.FTN$source (base-fiber ?j)) (arity-fiber ?j))
             (= (SET.FTN$target (base-fiber ?j)) set.obj$object)
             (SET.FTN$restriction (base-fiber ?j) base)))

(9) (SET.FTN$function set-fiber)
    (= (SET.FTN$source set-fiber) set.obj$object)
    (= (SET.FTN$target set-fiber) SET.FTN$function)
    (forall (?j (set.obj$object ?j))
        (and (= (SET.FTN$source (set-fiber ?j)) (arity-fiber ?j))
```

```
(= (SET.FTN$target (set-fiber ?j)) set.mor$morphism)
(= (SET.FTN$composition [(set-fiber ?j) set.mor$source]) ?j)
(= (SET.FTN$composition [(set-fiber ?j) set.mor$target])
   (SET.FTN$composition [(base-fiber ?j) set$power]))
(SET.FTN$restriction (set-fiber ?j) set)))
```

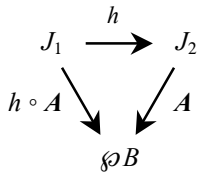


Diagram 6: Inverse Image – abstract

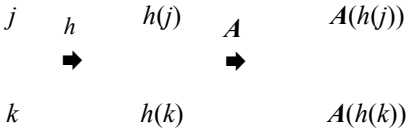


Figure 18: Inverse Image – details

Any set function $h : J_1 \to J_2$ defines by composition an *inverse image* operator $h^{-1} = \mathit{art}(h) : \mathit{art}(J_2) \to \mathit{art}(J_1)$ (Figure 18) that maps any arity $\boldsymbol{A} : J_2 \to \wp B$ to the arity $h \circ \boldsymbol{A} : J_1 \to \wp B$, whose index is (of course) the source set J_1, whose set function is the composition of h with the set function of $\boldsymbol{A}$.

```
(10) (SET.FTN$function inverse-image)
     (= (SET.FTN$source inverse-image) set.mor$morphism)
     (= (SET.FTN$target inverse-image) SET.FTN$function)
     (forall (?h (set.mor$morphism ?h))
         (and (= (SET.FTN$source (inverse-image ?h)) (arity-fiber (set.mor$target ?h)))
              (= (SET.FTN$target (inverse-image ?h)) (arity-fiber (set.mor$source ?h)))
              (= (SET.FTN$composition [(inverse-image ?h) (base-fiber (set.mor$source ?h))])
                 ((SET.FTN$constant
                     [(arity-fiber (set.mor$target ?h)) (arity-fiber (set.mor$source ?h))])
                        (set.mor$source ?h)))
              (forall (?a ((arity-fiber (set.mor$target ?h)) ?a))
                  (= ((set-fiber (set.mor$source ?h)) ((inverse-image ?h) ?a))
                     (set.mor$composition [?h ((set-fiber (set.mor$target ?h)) ?a)]))))))
```

A set function and an arity are *mixed composable* when the target of the first is equal to the index of the second. The *mixed composition* of such a composable pair $h : J_1 \to J_2$ and $\boldsymbol{A} : J_2 \to \wp B$ is defined in terms of the composition of h with the set function of $\boldsymbol{A}$.

```
(11) (SET.LIM.PBK$opspan mixed-composable-opspan)
     (= (SET.LIM.PBK$class1 mixed-composable-opspan) set.mor$morphism)
     (= (SET.LIM.PBK$class2 mixed-composable-opspan) arity)
     (= (SET.LIM.PBK$opvertex mixed-composable-opspan) set.obj$object)
     (= (SET.LIM.PBK$opfirst mixed-composable-opspan) set.mor$target)
     (= (SET.LIM.PBK$opsecond mixed-composable-opspan) index)

(12) (REL$relation mixed-composable)
     (= (REL$class1 mixed-composable) set.mor$morphism)
     (= (REL$class2 mixed-composable) arity)
     (= (REL$extent mixed-composable) (SET.LIM.PBK$pullback mixed-composable-opspan))

(13) (SET.FTN$function mixed-composition)
     (= (SET.FTN$source mixed-composition) (SET.LIM.PBK$pullback mixed-composable-opspan))
     (= (SET.FTN$target composition) arity)
     (forall (?h (set.mor$morphism ?h) ?a (arity ?a)
             (mixed-composable ?h ?a))
         (and (= (index (mixed-composition [?h ?a])) (set.mor$source ?h))
              (= (base (mixed-composition [?h ?a])) (base ?a))
              (= (set (mixed-composition [?h ?a]))
                 (set.mor$composition [?h (set ?a)])))))
```

Arity Morphisms

`set.dgm.art.mor`

Arities are connected by morphisms. Although an arity is a special case of a tuple, the morphism of arities defined here is not a special case of a tuple morphism, whether strict, lax or colax, since the base varies.

A *morphism of arities* $\mathsf{F} = \langle \mathit{ind}(\mathsf{F}), \mathit{base}(\mathsf{F}) \rangle : \boldsymbol{A}_1 \to \boldsymbol{A}_2$ (Diagram 7) consists of

– an index function $\mathit{ind}(\mathsf{F}) : \mathit{ind}(\boldsymbol{A}_1) \to \mathit{ind}(\boldsymbol{A}_2)$ and

– a base function $\mathit{base}(\mathsf{F}) : \mathit{base}(\boldsymbol{A}_1) \to \mathit{base}(\boldsymbol{A}_2)$

for which Diagram 7 commutes:

$$\mathit{ind}(\mathsf{F}) \cdot \mathit{set}(\boldsymbol{A}_2) = \mathit{set}(\boldsymbol{A}_1) \cdot \wp\, \mathit{base}(\mathsf{F}).$$

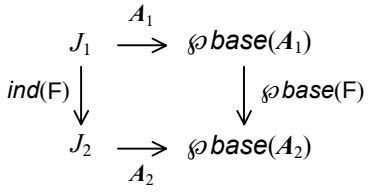


Diagram 7: Arity Morphism Constraint

Pointwise this state that $\wp\, \mathit{base}(\mathsf{F})(\boldsymbol{A}_1(n)) = \boldsymbol{A}_2(\mathit{ind}(\mathsf{F})(n))$ for every source index $n \in \mathit{ind}(\boldsymbol{A}_1)$.

This implies that for any pair (n, x), where $n \in \mathit{ind}(\boldsymbol{A}_1)$ is a source index and $x \in \boldsymbol{A}_1(n)$ is an element in the n^{th} component subset $\boldsymbol{A}_1(n) \subseteq \mathit{base}(\boldsymbol{A}_1)$, the pair $(m, y) = (\mathit{ind}(\mathsf{F})(n), \mathit{base}(\mathsf{F})(x))$ consists of a target index $m \in \mathit{ind}(\boldsymbol{A}_2)$ and an element $y \in \boldsymbol{A}_2(m)$ in the m^{th} component subset $\boldsymbol{A}_2(m) \subseteq \mathit{base}(\boldsymbol{A}_2)$.

```
(1) (SET$class morphism)

(2) (SET.FTN$function source)
    (= (SET.FTN$source source) morphism)
    (= (SET.FTN$target source) set.dgm.art$arity)

(3) (SET.FTN$function target)
    (= (SET.FTN$source target) morphism)
    (= (SET.FTN$target target) set.dgm.art$arity)

(4) (SET.FTN$function index)
    (= (SET.FTN$source index) morphism)
    (= (SET.FTN$target index) set.mor$morphism)
    (= (SET.FTN$composition [index set.mor$source])
       (SET.FTN$composition [source set.dgm.art$index]))
    (= (SET.FTN$composition [index set.mor$target])
       (SET.FTN$composition [target set.dgm.art$index]))

(5) (SET.FTN$function base)
    (= (SET.FTN$source base) morphism)
    (= (SET.FTN$target base) set.mor$morphism)
    (= (SET.FTN$composition [base set.mor$source])
       (SET.FTN$composition [source set.dgm.art$base]))
    (= (SET.FTN$composition [base set.mor$target])
       (SET.FTN$composition [target set.dgm.art$base]))

(6) (forall (?f (morphism ?f))
        (= (set.mor$composition [(index ?f) (set.dgm.art$set (target ?h))])
           (set.mor$composition [(set.dgm.art$set (source ?h)) (set.mor$power (base ?h))])))
```

Associated with any morphism of arities F is a *quartet* $\mathit{qtt}(\mathsf{F})$.

```
(7) (SET.FTN$function quartet)
    (= (SET.FTN$source quartet) morphism)
    (= (SET.FTN$target quartet) set.qtt$quartet)
    (= (SET.FTN$composition [quartet set.qtt$horizontal-source])
       (SET.FTN$composition [source set.dgm.art$set]))
    (= (SET.FTN$composition [quartet set.qtt$horizontal-target])
       (SET.FTN$composition [target set.dgm.art$set]))
    (= (SET.FTN$composition [quartet set.qtt$vertical-source]) index)
    (= (SET.FTN$composition [quartet set.qtt$vertical-target])
       (SET.FTN$composition [base set.mor$power]))
```

Two arity morphisms are *composable* when the target of the first is equal to the source of the second. The *composition* of two composable arity morphisms $\mathsf{F} = \langle \mathit{ind}(\mathsf{F}), \mathit{base}(\mathsf{F}) \rangle : \boldsymbol{A}_1 \to \boldsymbol{A}_2$ and $\mathsf{G} = \langle \mathit{ind}(\mathsf{G}), \mathit{base}(\mathsf{G}) \rangle : \boldsymbol{A}_2 \to \boldsymbol{A}_3$ is defined in terms of the composition of their component functions.

```
 (8) (SET.LIM.PBK$opspan composable-opspan)
     (= (SET.LIM.PBK$class1 composable-opspan) morphism)
     (= (SET.LIM.PBK$class2 composable-opspan) morphism)
     (= (SET.LIM.PBK$opvertex composable-opspan) set.dgm.art$arity)
     (= (SET.LIM.PBK$opfirst composable-opspan) target)
     (= (SET.LIM.PBK$opsecond composable-opspan) source)

 (9) (REL$relation composable)
     (= (REL$class1 composable) morphism)
     (= (REL$class2 composable) morphism)
     (= (REL$extent composable) (SET.LIM.PBK$pullback composable-opspan))

(10) (SET.FTN$function composition)
     (= (SET.FTN$source composition) (SET.LIM.PBK$pullback composable-opspan))
     (= (SET.FTN$target composition) morphism)
     (forall (?f (morphism ?f) ?g (morphism ?g)
             (composable ?f ?g))
         (and (= (source (composition [?f ?g])) (source ?f))
              (= (target (composition [?f ?g])) (target ?g))
              (= (index (composition [?f ?g]))
                 (set.ftn$composition [(index ?f) (index ?g)]))
              (= (base (composition [?f ?g]))
                 (set.ftn$composition [(base ?f) (base ?g)])))))
```

Composition satisfies the usual *associative law*.

```
     (forall (?f (morphism ?f)
              ?g (morphism ?g)
              ?h (morphism ?h)
              (composable ?f ?g) (composable ?g ?h))
         (= (composition [?f (composition [?g ?h])])
            (composition [(composition [?f ?g]) ?h])))
```

For any set arity $\boldsymbol{A} : J \to \wp B$, there is an *identity* arity morphism.

```
(11) (SET.FTN$function identity)
     (= (SET.FTN$source identity) set.dgm.art$arity)
     (= (SET.FTN$target identity) morphism)
     (forall (?a (set.dgm.art$arity ?a))
         (and (= (source (identity ?a)) ?a)
              (= (target (identity ?a)) ?a)
              (= (index (identity ?a)) (set.mor$identity (set.dgm.art$index ?a)))
              (= (base (identity ?a)) (set.mor$identity (set.dgm.art$base ?a)))))
```

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

```
     (forall (?f (morphism ?f))
         (and (= (composition [(identity (source ?f)) ?f]) ?f)
              (= (composition [?f (identity (target ?f))]) ?f)))
```

Pairs

`set.dgm.pr`

A *pair* (of sets) is the appropriate base diagram for a binary coproduct. Each pair consists of a pair of sets called *set1* and *set2*. We use either the generic term '`diagram`' or the specific term '`pair`' to denote the *pair* class. Pairs are determined by their two component sets. We connect this specific terminology to the generic fiber terminology for set diagrams.

```
(1) (SET$class diagram)
    (SET$class pair)
    (= pair diagram)
    (= diagram (set.dgm$diagram-fiber gph$two))

(2) (SET.FTN$function set1)
    (= (SET.FTN$source set1) diagram)
    (= (SET.FTN$target set1) set.obj$object)
    (forall (?p (diagram ?p))
        (= set1 (((set.dgm$set-fiber gph$two) ?p) 1)))

(3) (SET.FTN$function set2)
    (= (SET.FTN$source set2) diagram)
    (= (SET.FTN$target set2) set.obj$object)
    (forall (?p (diagram ?p))
        (= set2 (((set.dgm$set-fiber gph$two) ?p) 2)))

(4) (forall (?p (diagram ?p) ?q (diagram ?q))
        (=> (and (= (set1 ?p) (set1 ?q))
                 (= (set2 ?p) (set2 ?q)))
            (= ?p ?q)))
```

Pair Morphisms

`set.dgm.pr.mor`

Set pairs are connected by morphisms. A morphism of set pairs consists of a pair of functions connecting the respective set components.

```
(1) (SET$class morphism)
    (= morphism (set.dgm.mor$morphism-fiber gph$two))

(2) (SET.FTN$function source)
    (= (SET.FTN$source source) morphism)
    (= (SET.FTN$target source) set.dgm.pr$diagram)

(3) (SET.FTN$function target)
    (= (SET.FTN$source target) morphism)
    (= (SET.FTN$target target) set.dgm.pr$diagram)

(4) (SET.FTN$function function1)
    (= (SET.FTN$source function1) morphism)
    (= (SET.FTN$target function1) set.mor$morphism)
    (= (SET.FTN$[function1 set.mor$source])
       (SET.FTN$[source set.dgm.pr$set1]))
    (= (SET.FTN$[function1 set.mor$target])
       (SET.FTN$[target set.dgm.pr$set1]))
    (forall (?f (morphism ?f))
        (= function1 (((set.dgm.mor$component-fiber gph$two) ?f) 1)))

(5) (SET.FTN$function function2)
    (= (SET.FTN$source function2) morphism)
    (= (SET.FTN$target function2) set.mor$morphism)
    (= (SET.FTN$[function2 set.mor$source])
       (SET.FTN$[source set.dgm.pr$set2]))
    (= (SET.FTN$[function2 set.mor$target])
       (SET.FTN$[target set.dgm.pr$set2]))
    (forall (?f (morphism ?f))
        (= function2 (((set.dgm.mor$component-fiber gph$two) ?f) 2)))
```

Parallel Pairs

set.dgm.ppr

A *parallel pair* (see Figure 19, where arrows denote set functions) is the appropriate base diagram for a coequalizer. Each parallel pair consists of a pair of set functions called *function1* and *function2* that share the same *origin* and *destination* sets. We use either the generic term ‘diagram’ or the specific term ‘parallel-pair’ to denote the *parallel pair* class. Parallel pairs are determined by their two component set functions. We connect this specific terminology to the generic fiber terminology for set diagrams.

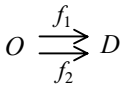


Figure 19: Parallel Pair

```
(1) (SET$class diagram)
    (SET$class parallel-pair)
    (= parallel-pair diagram)
    (= diagram (set.dgm$diagram-fiber gph$parallel-pair))

(2) (SET.FTN$function origin)
    (= (SET.FTN$source origin) diagram)
    (= (SET.FTN$target origin) set.obj$object)
    (forall (?pp (diagram ?pp))
        (= origin (((set.dgm$set-fiber gph$parallel-pair) ?pp) 1)))

(3) (SET.FTN$function destination)
    (= (SET.FTN$source destination) diagram)
    (= (SET.FTN$target destination) set.obj$object)
    (forall (?pp (diagram ?pp))
        (= destination (((set.dgm$set-fiber gph$parallel-pair) ?pp) 2)))

(4) (SET.FTN$function function1)
    (= (SET.FTN$source function1) diagram)
    (= (SET.FTN$target function1) set.mor$morphism)
    (= (SET.FTN$composition [function1 set.mor$source]) origin)
    (= (SET.FTN$composition [function1 set.mor$target]) destination)
    (forall (?pp (diagram ?pp))
        (= function1 (((set.dgm$function-fiber gph$parallel-pair) ?pp) 1)))

(5) (SET.FTN$function function2)
    (= (SET.FTN$source function2) diagram)
    (= (SET.FTN$target function2) set.mor$morphism)
    (= (SET.FTN$composition [function2 set.mor$source]) origin)
    (= (SET.FTN$composition [function2 set.mor$target]) destination)
    (forall (?pp (diagram ?pp))
        (= function2 (((set.dgm$function-fiber gph$parallel-pair) ?pp) 2)))

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

The information in a parallel pair of set functions expresses a set *endorelation* based at the destination set of the parallel pair.

```
(7) (SET.FTN$function endorelation)
    (= (SET.FTN$source endorelation) diagram)
    (= (SET.FTN$target endorelation) set.col.endo$endorelation)
    (= (SET.FTN$composition [endorelation set.col.endo$set]) destination)
    (forall (?d (diagram ?d)
             ?b1 ((destination ?d) ?b1)
             ?b2 ((destination ?d) ?b2))
        (<=> ((set.col.endo$element (endorelation ?d)) [?b1 ?b2])
             (exists (?a ((origin ?d) ?a))
                 (and (= ?b1 ((function1 ?d) ?a))
                      (= ?b2 ((function2 ?d) ?a)))))))
```

The information in a parallel pair of set functions expresses a set *subobject* based at the origin set of the parallel pair.

```
(8) (SET.FTN$function subobject)
```

```
    (= (SET.FTN$source subobject) diagram)
    (= (SET.FTN$target subobject) set.lim.sub$subset)
    (= (SET.FTN$composition [subobject set.lim.sub$set]) origin)
    (forall (?d (diagram ?d)
             ?a ((origin ?d) ?a))
        (<=> ((set.lim.sub$subset (subobject ?d)) ?a)
             (= ((function1 ?d) ?a)
                ((function2 ?d) ?a))))
```

Parallel Pair Morphisms

`set.dgm.ppr.mor`

Set parallel pairs are connected by morphisms. A morphism of set parallel pairs consists of a pair of functions connecting the respective set components.

```
(1) (SET$class morphism)
    (= morphism (set.dgm.mor$morphism-fiber gph$parallel-pair))

(2) (SET.FTN$function source)
    (= (SET.FTN$source source) morphism)
    (= (SET.FTN$target source) set.dgm.ppr$diagram)

(3) (SET.FTN$function target)
    (= (SET.FTN$source target) morphism)
    (= (SET.FTN$target target) set.dgm.ppr$diagram)

(4) (SET.FTN$function origin)
    (= (SET.FTN$source origin) morphism)
    (= (SET.FTN$target origin) set.mor$morphism)
    (= (SET.FTN$[origin set.mor$source])
       (SET.FTN$[source set.dgm.ppr$origin]))
    (= (SET.FTN$[origin set.mor$target])
       (SET.FTN$[target set.dgm.ppr$origin]))
    (forall (?f (morphism ?f))
        (= origin (((set.dgm.mor$component-fiber gph$parallel-pair) ?f) 1)))

(5) (SET.FTN$function destination)
    (= (SET.FTN$source destination) morphism)
    (= (SET.FTN$target destination) set.mor$morphism)
    (= (SET.FTN$[destination set.mor$source])
       (SET.FTN$[source set.dgm.ppr$destination]))
    (= (SET.FTN$[destination set.mor$target])
       (SET.FTN$[target set.dgm.ppr$destination]))
    (forall (?f (morphism ?f))
        (= destination (((set.dgm.mor$component-fiber gph$parallel-pair) ?f) 2)))
```

Spans

set.dgm.spn

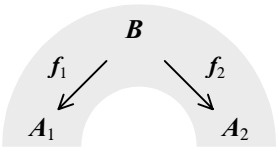


Figure 20: Span

A *span* (Figure 20) is the appropriate base diagram for a pushout. Each span consists of a pair of set functions called *first* and *second*. These are required to have a common source set called the *vertex*. We use either the generic term '`diagram`' or the specific term '`span`' to denote the *span* class. A span is the special case of a general diagram whose shape is the graph that is also named *span*. Spans are determined by their pair of component set functions. We connect this specific terminology to the generic fiber terminology for set diagrams.

```
(1) (SET$class diagram)
    (SET$class span)
    (= span diagram)
    (= diagram (set.dgm$diagram-fiber gph$span))

(2) (SET.FTN$function set1)
    (= (SET.FTN$source set1) diagram)
    (= (SET.FTN$target set1) set.obj$object)
    (forall (?r (diagram ?r))
        (= set1 (((set.dgm$set-fiber gph$span) ?r) 1)))

(3) (SET.FTN$function set2)
    (= (SET.FTN$source set2) diagram)
    (= (SET.FTN$target set2) set.obj$object)
    (forall (?r (diagram ?r))
        (= set2 (((set.dgm$set-fiber gph$span) ?r) 2)))

(4) (SET.FTN$function vertex)
    (= (SET.FTN$source vertex) diagram)
    (= (SET.FTN$target vertex) set.obj$object)
    (forall (?r (diagram ?r))
        (= vertex (((set.dgm$set-fiber gph$span) ?r) 3)))

(5) (SET.FTN$function first)
    (= (SET.FTN$source first) diagram)
    (= (SET.FTN$target first) set.mor$morphism)
    (= (SET.FTN$composition [first set.mor$source]) vertex)
    (= (SET.FTN$composition [first set.mor$target]) set1)
    (forall (?r (diagram ?r))
        (= first (((set.dgm$function-fiber gph$span) ?r) 1)))

(6) (SET.FTN$function second)
    (= (SET.FTN$source second) diagram)
    (= (SET.FTN$target second) set.mor$morphism)
    (= (SET.FTN$composition [second set.mor$source]) vertex)
    (= (SET.FTN$composition [second set.mor$target]) set2)
    (forall (?r (diagram ?r))
        (= second (((set.dgm$function-fiber gph$span) ?r) 2)))

(7) (forall (?d1 (diagram ?d1) ?d2 (diagram ?d2))
        (=> (and (= (first ?d1) (first ?d2))
                 (= (second ?d1) (second ?d2)))
            (= ?d1 ?d2)))
```

Every span has an opposite.

```
(8) (SET.FTN$function opposite)
    (= (SET.FTN$source opposite) span)
    (= (SET.FTN$target opposite) span)
    (= (SET.FTN$composition [opposite set1]) set2)
    (= (SET.FTN$composition [opposite set2]) set1)
    (= (SET.FTN$composition [opposite vertex]) vertex)
    (= (SET.FTN$composition [opposite first]) second)
    (= (SET.FTN$composition [opposite second]) first)
```

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

```
(9) (= (SET.FTN$composition [opposite opposite])
       (SET.FTN$identity span))
```

The *pair* of target sets (suffixing discrete diagram) underlying any span is named. This construction is derived from the fact that the pair shape is a subshape of the span shape.

```
(10) (SET.FTN$function pair)
     (= (SET.FTN$source pair) diagram)
     (= (SET.FTN$target pair) set.dgm.pr$diagram)
     (= (SET.FTN$composition [pair set.dgm.pr$set1]) set1)
     (= (SET.FTN$composition [pair set.dgm.pr$set2]) set2)
```

The *parallel pair* function maps a span of set functions to the associated specific parallel pair of set functions (see Figure 21, where arrows denote set functions). These functions are the composite of the first and second functions of the span with the coproduct injections for the specific binary coproduct of the pair of component sets in the span. The coequalizer and canon of the associated parallel pair will be used to define the pushout.

For the convenience of the implementer, we also provide a *standard parallel pair* that uses the standard coproduct or disjoint union of the pair of sets of a span.

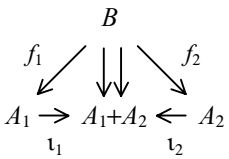


Figure 21: Coequalizer Diagram

```
(11) (SET.FTN$function parallel-pair)
     (= (SET.FTN$source parallel-pair) diagram)
     (= (SET.FTN$target parallel-pair) set.dgm.ppr$diagram)
     (= (SET.FTN$composition [parallel-pair set.dgm.ppr$origin]) vertex)
     (= (SET.FTN$composition [parallel-pair set.dgm.ppr$destination])
        (SET.FTN$composition [pair set.col.coprd2$binary-coproduct]))
     (forall (?d (diagram ?d))
         (and (= (set.dgm.ppr$function1 (parallel-pair ?d))
                 (set.mor$composition [(first ?d) (set.col.coprd2$injection1 (pair ?d))]))
              (= (set.dgm.ppr$function2 (parallel-pair ?d))
                 (set.mor$composition [(second ?d) (set.col.coprd2$injection2 (pair ?d))])))))

(12) (SET.FTN$function standard-parallel-pair)
     (= (SET.FTN$source standard-parallel-pair) diagram)
     (= (SET.FTN$target standard-parallel-pair) set.dgm.ppr$diagram)
     (= (SET.FTN$composition [standard-parallel-pair set.dgm.ppr$origin]) vertex)
     (= (SET.FTN$composition [standard-parallel-pair set.dgm.ppr$destination])
        (SET.FTN$composition [pair set.col.coprd2$standard-binary-coproduct]))
     (forall (?d (diagram ?d))
         (and (= (set.dgm.ppr$function1 (standard-parallel-pair ?d))
                 (set.mor$composition
                     [(first ?d) (set.col.coprd2$standard-injection1 (pair ?d))]))
              (= (set.dgm.ppr$function2 (standard-parallel-pair ?d))
                 (set.mor$composition
                     [(second ?d) (set.col.coprd2$standard-injection2 (pair ?d))])))))
```

Span Morphisms

`set.dgm.spn.mor`

Set spans are connected by morphisms. A morphism of set spans consists of a triple of functions connecting the respective set components.

```
(1) (SET$class morphism)
    (= morphism (set.dgm.mor$morphism-fiber gph$span))

(2) (SET.FTN$function source)
    (= (SET.FTN$source source) morphism)
    (= (SET.FTN$target source) set.dgm.spn$diagram)

(3) (SET.FTN$function target)
    (= (SET.FTN$source target) morphism)
    (= (SET.FTN$target target) set.dgm.spn$diagram)

(4) (SET.FTN$function function1)
    (= (SET.FTN$source function1) morphism)
    (= (SET.FTN$target function1) set.mor$morphism)
```

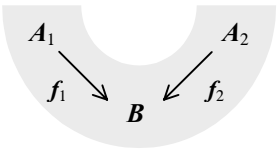
```
    (= (SET.FTN$[function1 set.mor$source])
       (SET.FTN$[source set.dgm.spn$set1]))
    (= (SET.FTN$[function1 set.mor$target])
       (SET.FTN$[target set.dgm.spn$set1]))
    (forall (?f (morphism ?f))
        (= function1 (((set.dgm.mor$component-fiber gph$span) ?f) 1)))

(5) (SET.FTN$function function2)
    (= (SET.FTN$source function2) morphism)
    (= (SET.FTN$target function2) set.mor$morphism)
    (= (SET.FTN$[function2 set.mor$source])
       (SET.FTN$[source set.dgm.spn$set2]))
    (= (SET.FTN$[function2 set.mor$target])
       (SET.FTN$[target set.dgm.spn$set2]))
    (forall (?f (morphism ?f))
        (= function2 (((set.dgm.mor$component-fiber gph$span) ?f) 2)))

(6) (SET.FTN$function vertex)
    (= (SET.FTN$source vertex) morphism)
    (= (SET.FTN$target vertex) set.mor$morphism)
    (= (SET.FTN$[vertex set.mor$source])
       (SET.FTN$[source set.dgm.spn$vertex]))
    (= (SET.FTN$[vertex set.mor$target])
       (SET.FTN$[target set.dgm.spn$vertex]))
    (forall (?f (morphism ?f))
        (= vertex (((set.dgm.mor$component-fiber gph$span) ?f) 3)))
```

Opspans

set.dgm.ospn

An *opspan* (Figure 22) is the appropriate base diagram for a pullback. Each opspan consists of a pair of set functions called *opfirst* and *opsecond*. These are required to have a common target set called the *opvertex*. We use either the generic term ‘diagram’ or the specific term ‘opspan’ to denote the *opspan* class. An opspan is the special case of a general diagram whose shape is the graph that is also named *opspan*. Opspans are determined by their pair of component set functions. We connect this specific terminology to the generic fiber terminology for set diagrams.

Figure 22: Opspan




```
(1) (SET$class diagram)
    (SET$class opspan)
    (= opspan diagram)
    (= diagram (set.dgm$diagram-fiber gph$opspan))

(2) (SET.FTN$function set1)
    (= (SET.FTN$source set1) diagram)
    (= (SET.FTN$target set1) set.obj$object)
    (forall (?s (diagram ?s))
        (= set1 (((set.dgm$set-fiber gph$opspan) ?s) 1)))

(3) (SET.FTN$function set2)
    (= (SET.FTN$source set2) diagram)
    (= (SET.FTN$target set2) set.obj$object)
    (forall (?s (diagram ?s))
        (= set2 (((set.dgm$set-fiber gph$opspan) ?s) 2)))

(4) (SET.FTN$function opvertex)
    (= (SET.FTN$source opvertex) diagram)
    (= (SET.FTN$target opvertex) set.obj$object)
    (forall (?s (diagram ?s))
        (= opvertex (((set.dgm$set-fiber gph$opspan) ?s) 3)))

(5) (SET.FTN$function opfirst)
    (= (SET.FTN$source opfirst) diagram)
    (= (SET.FTN$target opfirst) set.mor$morphism)
    (= (SET.FTN$composition [opfirst set.mor$source]) set1)
    (= (SET.FTN$composition [opfirst set.mor$target]) opvertex)
    (forall (?s (diagram ?s))
        (= opfirst (((set.dgm$function-fiber gph$opspan) ?s) 1)))

(6) (SET.FTN$function opsecond)
    (= (SET.FTN$source opsecond) diagram)
    (= (SET.FTN$target opsecond) set.mor$morphism)
    (= (SET.FTN$composition [opsecond set.mor$source]) set2)
    (= (SET.FTN$composition [opsecond set.mor$target]) opvertex)
    (forall (?s (diagram ?s))
        (= opsecond (((set.dgm$function-fiber gph$opspan) ?s) 2)))

(7) (forall (?d1 (diagram ?d1) ?d2 (diagram ?d2))
        (=> (and (= (opfirst ?d1) (opfirst ?d2))
                 (= (opsecond ?d1) (opsecond ?d2)))
            (= ?d1 ?d2)))
```

Every opspan has an opposite.

```
(8) (SET.FTN$function opposite)
    (= (SET.FTN$source opposite) opspan)
    (= (SET.FTN$target opposite) opspan)
    (= (SET.FTN$composition [opposite set1]) set2)
    (= (SET.FTN$composition [opposite set2]) set1)
    (= (SET.FTN$composition [opposite opvertex]) opvertex)
    (= (SET.FTN$composition [opposite opfirst]) opsecond)
    (= (SET.FTN$composition [opposite opsecond]) opfirst)
```

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

```
(9) (= (SET.FTN$composition [opposite opposite])
       (SET.FTN$identity opspan))
```

The *pair* of source sets (prefixing discrete diagram) overlying any opspan is named. This construction is derived from the fact that the pair shape is a subshape of the opspan shape.

```
(10) (SET.FTN$function pair)
     (= (SET.FTN$source pair) diagram)
     (= (SET.FTN$target pair) set.lim.prd2$diagram)
     (= (SET.FTN$composition [pair set.lim.prd2$set1]) set1)
     (= (SET.FTN$composition [pair set.lim.prd2$set2]) set2)
```

The *parallel pair* function maps an opspan of set functions to the associated (`set.lim.equ`) parallel pair of set functions (see Figure 23, where arrows denote set functions). These functions are the composite of the product projections for the binary product of the pair of component sets in the opspan with the opfirst and opsecond functions of the opspan. The equalizer and canon of the associated parallel pair will be used to define the pullback.

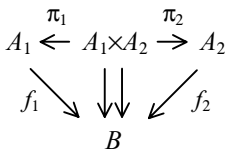


Figure 23: Equalizer Diagram

For the convenience of the implementer, we also provide a *standard parallel pair* that uses the standard product or Cartesian product of the pair of sets of an opspan.

```
(11) (SET.FTN$function parallel-pair)
     (= (SET.FTN$source parallel-pair) diagram)
     (= (SET.FTN$target parallel-pair) set.dgm.ppr$diagram)
     (= (SET.FTN$composition [parallel-pair set.dgm.ppr$origin])
        (SET.FTN$composition [pair set.lim.prd2$binary-product]))
     (= (SET.FTN$composition [parallel-pair set.dgm.ppr$destination]) opvertex)
     (forall (?d (diagram ?d))
         (and (= (set.dgm.ppr$function1 (parallel-pair ?d))
                 (set.mor$composition
                     [(set.lim.prd2$projection1 (pair ?d)) (opfirst ?d)]))
              (= (set.dgm.ppr$function2 (parallel-pair ?d))
                 (set.mor$composition
                     [(set.lim.prd2$$projection2 (pair ?d)) (opsecond ?d)])))))

(12) (SET.FTN$function standard-parallel-pair)
     (= (SET.FTN$source standard-parallel-pair) diagram)
     (= (SET.FTN$target standard-parallel-pair) set.dgm.ppr$diagram)
     (= (SET.FTN$composition [standard-parallel-pair set.dgm.ppr$origin])
        (SET.FTN$composition [pair set.lim.prd2$standard-binary-product]))
     (= (SET.FTN$composition [standard-parallel-pair set.dgm.ppr$destination]) opvertex)
     (forall (?d (diagram ?d))
         (and (= (set.dgm.ppr$function1 (standard-parallel-pair ?d))
                 (set.mor$composition
                     [(set.lim.prd2$standard-projection1 (pair ?d)) (opfirst ?d)]))
              (= (set.dgm.ppr$function2 (standard-parallel-pair ?d))
                 (set.mor$composition
                     [(set.lim.prd2$standard-projection2 (pair ?d)) (opsecond ?d)])))))
```

Opspan Morphisms

`set.dgm.ospn.mor`

Set opspans are connected by morphisms. A morphism of set opspans consists of a triple of functions connecting the respective set components.

```
(1) (SET$class morphism)
    (= morphism (set.dgm.mor$morphism-fiber gph$opspan))

(2) (SET.FTN$function source)
    (= (SET.FTN$source source) morphism)
    (= (SET.FTN$target source) set.dgm.ospn$diagram)

(3) (SET.FTN$function target)
    (= (SET.FTN$source target) morphism)
    (= (SET.FTN$target target) set.dgm.ospn$diagram)

(4) (SET.FTN$function function1)
```

```
    (= (SET.FTN$source function1) morphism)
    (= (SET.FTN$target function1) set.mor$morphism)
    (= (SET.FTN$[function1 set.mor$source])
       (SET.FTN$[source set.dgm.ospn$set1]))
    (= (SET.FTN$[function1 set.mor$target])
       (SET.FTN$[target set.dgm.ospn$set1]))
    (forall (?f (morphism ?f))
        (= function1 (((set.dgm.mor$component-fiber gph$opspan) ?f) 1)))

(5) (SET.FTN$function function2)
    (= (SET.FTN$source function2) morphism)
    (= (SET.FTN$target function2) set.mor$morphism)
    (= (SET.FTN$[function2 set.mor$source])
       (SET.FTN$[source set.dgm.ospn$set2]))
    (= (SET.FTN$[function2 set.mor$target])
       (SET.FTN$[target set.dgm.ospn$set2]))
    (forall (?f (morphism ?f))
        (= function2 (((set.dgm.mor$component-fiber gph$opspan) ?f) 2)))

(6) (SET.FTN$function opvertex)
    (= (SET.FTN$source opvertex) morphism)
    (= (SET.FTN$target opvertex) set.mor$morphism)
    (= (SET.FTN$[opvertex set.mor$source])
       (SET.FTN$[source set.dgm.ospn$opvertex]))
    (= (SET.FTN$[opvertex set.mor$target])
       (SET.FTN$[target set.dgm.ospn$opvertex]))
    (forall (?f (morphism ?f))
        (= opvertex (((set.dgm.mor$component-fiber gph$opspan) ?f) 3)))
```

Limits of Diagrams

`set.lim`

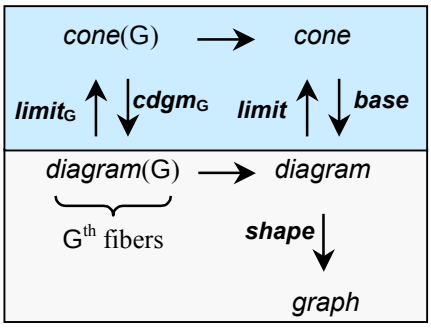


Diagram 8: Diagrams, Cocones and Fibers

Here we present axioms that make Set, the category of sets and set functions, complete. We assert the existence of terminal sets, products of collections of sets, equalizers of parallel pairs of set functions, and pullbacks of set opspans. All are defined to be specific sets; that is, sets specified by the implementer. In order to prevent clash of common terminology, the axiomatization of binary products, equalizers and pullbacks are put into sub-namespaces. This common terminology has been abstracted into a general formulation of limits. The *diagrams* and *limits* in the sub-namespaces are denoted by both generic and specific terminology. A *limit* is the vertex of a limiting cone of a (base) diagram of a certain shape. The general formulation (Diagram 8) presented first, is parameterized by the shape (graph) of the overlying base diagram for a limit. The shape of the overlying base diagram uses terminology from the graph namespace of the IFF Lower Core Ontology (IFF-LCO). The limits of a particular shape diagram are related to the general formulation by an axiomatization for limit fibers. Many of the other complete categories axiomatized in the IFF lower metalevel make use of Set limits via there the component functors and natural transformations that describe their objects and morphisms.

Limits of sets are the most basic. They underlie the limit construction for any complete category that is axiomatized in the IFF lower metalevel. In particular, they are used in the IFF Ontology (meta) Ontology (IFF-OO) for constructing the *instance pole* of object-level ontologies. As demonstrated below, since Set has all products and equalizers, it has all limits – it is complete. There are three expressions for completeness that range along a spectrum of definiteness:

- (weak expression) an axiomatized assertion of existence,
- (moderate expression) the description of a map from any diagram to a limiting cone, with axiomatized assertions that this is a limit for the diagram, and
- (strong expression) an explicit description with axioms of one particular limit.

The strong expression is most concrete, but perhaps too concrete in that it does not allow choices for implementers. In this namespace and throughout the IFF in general, we describe limits using the moderate expression; That is, not only do we provide an axiomatized assertion of the existence of limits, but we also claim, and require in any implementation, the existence of a map from any diagram to a (canonical) limit. Therefore, while we require a canonical limit to be described, we leave it up to the implementer how to make that canonical description. Of course, for any particular diagram all such limits must be unique up to isomorphism.

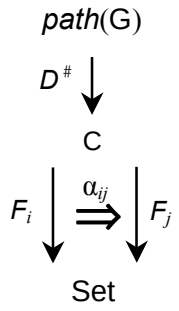


Figure 24: Component Diagrams and Diagram Morphisms

For any complete category C axiomatized in a namespace of the IFF lower metalevel and for any diagram D in C of (small) shape diagram G, it is often the case that the limiting cone of D is described implicitly in terms of the limits of certain diagrams in Set. An illustrative example is the category C = Hypergraph. Figure 24 abstractly describes this situation. Here we assume that objects and morphisms of C are axiomatized (usually uniquely) in terms of certain component structures describe abstractly by component functors (…, F_i, …, F_j, etc.) and natural transformations (…, α_{ij}, etc.) with target category Set. Since Set is complete as axiomatized in this namespace, there are maps from the diagrams, $D^{\#} \circ F_i$, …, $D^{\#} \circ F_j$, etc., to Set limiting cones, and there are maps from the diagram morphisms, …, $D^{\#} \circ \alpha_{ij}$, etc., to Set limiting cone morphisms. These maps, to Set limiting cones and limiting cone morphisms, are then used to describe the appropriate map from D to a limiting cone in C.

Table 5: Shapes for Diagrams

empty (terminal)	*two* (binary product)	*parallel pair* (equalizer)	*opspan* (pullback)

In the sections below, we axiomatize limits for finite diagrams $D : G \to |Set|$ of the following finite shape graphs G (Table 5): (i) *empty*, (ii) *two*, (iii) *parallel-pair* and (iv) *span*. These limits are called (i) the terminal set, (ii) a binary product, (iii) an equalizer and (iv) a pullback. The sections for binary products and subsets are crucial, since equalizers, pullbacks and all other limits depend upon them.

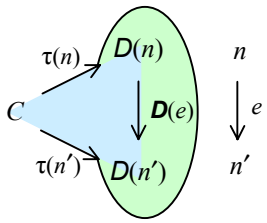


Diagram 9: Cone

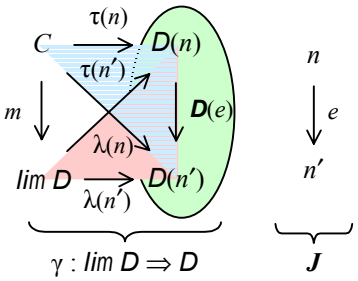


Diagram 10: Limiting Cone

A Set-valued *cone* (Diagram 9) consists of an overlying Set-valued *diagram*, an *vertex* set, and a collection of *component* set functions indexed by the nodes in the shape of the diagram. A cone is situated under its diagram. The (external) component set functions of the cone form commutative diagrams with the (internal) set functions of the overlying diagram.

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

(2) (KIF$function cone-diagram)
    (= (KIF$source cone-diagram) cone)
    (= (KIF$target cone-diagram) set.dgm$diagram)

(3) (KIF$function vertex)
    (= (KIF$source vertex) cone)
    (= (KIF$target vertex) set.obj$object)

(4) (KIF$function component)
    (= (KIF$source component) cone)
    (= (KIF$target component) SET.FTN$function)
    (forall (?s (cone ?s))
        (and (= (SET.FTN$source (component ?s))
                (gph.obj$node (set.dgm$shape (cone-diagram ?s))))
             (= (SET.FTN$target (component ?s)) set.mor$morphism)
             (= (SET.FTN$composition [(component ?s) set.mor$source])
                (SET.FTN$constant
                    [(gph.obj$node (set.dgm$shape (cone-diagram ?s))) set.obj$object]
                        (vertex ?s)))
             (= (SET.FTN$composition [(component ?s) set.mor$target])
                (set.dgm$set (cone-diagram ?s)))
             (forall (?e ((gph.obj$edge (set.dgm$shape (cone-diagram ?s))) ?e))
                 (= (set.mor$composition
                        [((component ?s) ((gph.obj$source (set.dgm$shape (cone-diagram ?s))) ?e))
                         ((set.dgm$function (cone-diagram ?s)) ?e)])
                    ((component ?s) ((gph.obj$target (set.dgm$shape (cone-diagram ?s))) ?e))))))
```

The *fiber* cone(G) for a (small) graph G is the class of all cones whose overlying diagram has shape G.

```
(5) (KIF$function diagram-shape)
    (= (KIF$source diagram-shape) cone)
    (= (KIF$target diagram-shape) gph.obj$object)
    (forall (?s (cone ?s))
        (= (diagram-shape ?s) (set.dgm$shape (cone-diagram ?s))))

(6) (KIF$function cone-fiber)
    (= (KIF$source cone-fiber) gph.obj$object))
    (= (KIF$target cone-fiber) SET$class)
    (= cone-fiber (KIF$fiber diagram-shape))

(7) (KIF$function cone-diagram-fiber)
    (= (KIF$source cone-diagram-fiber) gph.obj$object)
    (= (KIF$target cone-diagram-fiber) SET.FTN$function)
    (forall (?g (gph.obj$object ?g))
        (and (= (SET.FTN$source (cone-diagram-fiber ?g)) (cone-fiber ?g))
             (= (SET.FTN$target (cone-diagram-fiber ?g)) (set.dgm$diagram-fiber ?g))
             (forall (?s ((cone-fiber ?g) ?s))
                 (= ((cone-diagram-fiber ?g) ?s) (cone-diagram ?s)))))

(8) (KIF$function vertex-fiber)
    (= (KIF$source vertex-fiber) gph.obj$object)
    (= (KIF$target vertex-fiber) SET.FTN$function)
    (forall (?g (gph.obj$object ?g))
        (and (= (SET.FTN$source (vertex-fiber ?g)) (cone-fiber ?g))
             (= (SET.FTN$target (vertex-fiber ?g)) set.obj$object)
             (forall (?s ((cone-fiber ?g) ?s))
                 (= ((vertex-fiber ?g) ?s) (vertex ?s)))))

(9) (KIF$function component-fiber)
    (= (KIF$source component-fiber) gph.obj$object)
    (= (KIF$target component-fiber) KIF$function)
    (forall (?g (gph.obj$object ?g))
        (and (= (KIF$source (component-fiber ?g)) (cone-fiber ?g))
             (= (KIF$target (component-fiber ?g)) SET.FTN$function)
             (forall (?s ((cone-fiber ?g) ?s))
                 (and (= (SET.FTN$source ((component-fiber ?g) ?s)) (gph.obj$node ?g))
                      (= (SET.FTN$target ((component-fiber ?g) ?s)) set.mor$morphism)
                      (= ((component-fiber ?g) ?s) (component ?s)))))))
```

The *limiting cone* function maps a generic diagram of sets and set functions to its limiting cone (Diagram 10). The vertex of the limiting cone is a specific *limit* set $\mathit{lim}\ \mathbf{D} = \mathit{lim}(\mathbf{D})$ represented by the *limit* function. It comes equipped with specific component *projection* (an abuse of the terminology for set products) set functions $\lambda(n) = \mathit{pr}(\mathbf{D})(n) : \mathit{lim}\ \mathbf{D} \to \mathbf{D}(n)$. The latter two terms have been created for convenience of reference. These three choice functions assert that a specific limiting cone, limit and collection of projections exist for any diagram. The requirement here is that the implementer should construct a limit for every diagram, but it does not require a particular limit to be used. This allows maximum flexibility for implementation. Of course, any chosen limit is unique up to isomorphism. The universality of this limit is expressed by axioms for the mediator function.

```
(10) (KIF$function limiting-cone)
     (= (KIF$source limiting-cone) set.dgm$diagram)
     (= (KIF$target limiting-cone) cone)
     (forall (?d (set.dgm$diagram ?d))
         (= (cone-diagram (limiting-cone ?d)) ?d))

(11) (KIF$function limit)
     (= (KIF$source limit) set.dgm$diagram)
     (= (KIF$target limit) set.obj$object)
     (forall (?d (set.dgm$diagram ?d))
         (= (limit ?d) (vertex (limiting-cone ?d))))

(12) (KIF$function projection)
     (= (KIF$source projection) set.dgm$diagram)
     (= (KIF$target projection) SET.FTN$function)
     (forall (?d (set.dgm$diagram ?d))
         (and (= (SET.FTN$source (projection ?d)) (gph.obj$node (set.dgm$shape ?d)))
              (= (SET.FTN$target (projection ?d)) set.mor$morphism)
              (= (SET.FTN$composition [(projection ?d) set.mor$source])
```

```
   (SET.FTN$constant [(gph.obj$node (set.dgm$shape ?d)) set.obj$object])
   (limit ?d)))
(= (SET.FTN$composition [(projection ?d) set.mor$target]) (set.dgm$set ?d))
(= (projection ?d) (component (limiting-cone ?d)))))
```

For the convenience of the implementer, we also provide a standard limit and a standard collection of limit projection functions that she may choose. Here we closely follow section V.2 of Mac Lane on "Limits by Products and Equalizers". The construction of the *standard limit* $\underline{lim}\, D$ and *standard projections* $\lambda_n : \underline{lim}\, D \to D_n$ for a generic diagram D of sets and set functions is made in two steps. We need to make use of two coproducts in the standard colimit construction.

- Let the following denote the standard product and product projections

  $\pi_n : \prod_{n \in node(G)} D_n = \prod tpl(D) \to D(n)$

  for the *tuple* (object function) $tpl(D) : node(G) \to obj(Set)$ of D regarded as a tuple.

- In addition, we will also need to use the product and product projections

  $\pi_e : \prod_{e \in edge(G)} D_{tgt(e)} = \prod(tgt(G) \cdot tpl(D)) \to D(tgt(e))$

  for the *target tuple* $tgt(G) \cdot tpl(D) : edge(G) \to node(G) \to obj(Set)$.

The standard limit and standard limit projection functions form a product cone over the tuple (discrete diagram) $tpl(D)$. Hence, there is a product mediator function $m : \underline{lim}\, D \to \prod_{n \in node(G)} D_n$ satisfying the constraints $m \cdot \pi_n = \lambda_n$ for all $n \in node(G)$. That is, for any $n \in node(G)$ and any element $x \in \underline{lim}\, D$ of the limit set, the element $\lambda_n(x) = \pi_n(m(x)) = \pi_n(x) = x_n$ is an element of the component set D_n, where π_n is the standard product projection. For any edge $e : j \to k$ in the shape graph G of D, each element of $x \in lim\, D$ satisfies the λ-cone constraint $D(e)(x_j) = x_k$. For $j = src(e)$ and $k = tgt(e)$, this amounts to requiring that x lie in the standard equalizer of the parallel pair (Diagram 11) of the two set functions

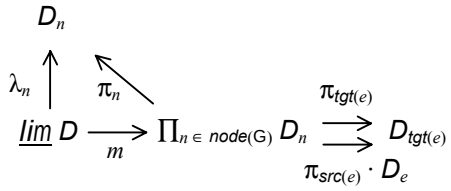


Diagram 11: Parallel Pair for edge e

$\pi_{tgt(e)} : \prod_{n \in node(G)} D_n \to D_{tgt(e)}$ and
$\pi_{src(e)} \cdot D_e : \prod_{n \in node(G)} D_n \to D_{src(e)} \to D_{tgt(e)}$.

In the standard limit construction (Diagram 12) we collect together all of the target domains $D_{tgt(e)}$ into the product $\prod_{e \in edge(G)} D_{tgt(e)}$. The following are the steps for computing the standard limit and its projections in the form of specification statements.

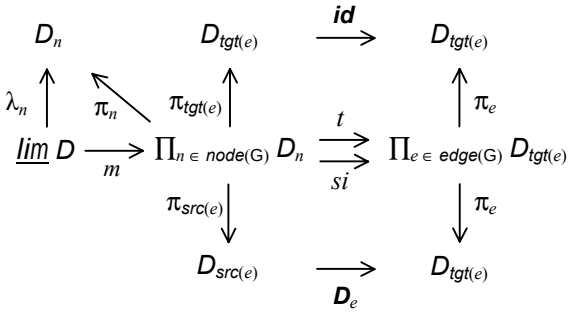


Diagram 12: Limits in terms of Equalizers and Products

First. Specify the *equalizer diagram* (parallel pair of set functions).

a. Specify the *target* function

   $t : \prod_{n \in node(G)} D_n = \prod tpl(D) \to \prod_{e \in edge(G)} D_{tgt(e)} = \prod(tgt(G) \cdot tpl(D))$

   as the product mediator for the *target cone*, whose components are the collection of set functions

   $\{\pi_{tgt(e)} : \prod_{n \in node(G)} D_n = \prod tpl(D) \to D_{tgt(e)} \mid e \in edge(G)\}$.

This is illustrated by the upper square in Diagram 12.

b. Specify the *source image* function

$si : \prod_{n \in node(\mathsf{G})} D_n = \prod tpl(D) \to \prod_{e \in edge(\mathsf{G})} D_{tgt(e)} = \prod(tgt(\mathsf{G}) \cdot tpl(D))$

as the product mediator (or tupling) for the for the *source image cone*, whose components are the collection of set functions

$\{\pi_{src(e)} \cdot \boldsymbol{D}_e : \prod_{n \in node(\mathsf{G})} D_n \to D_{src(e)} \to D_{tgt(e)} \mid e \in edge(\mathsf{G})\}$.

This is illustrated by the lower square in Diagram 12.

c. Form the parallel pair (equalizer diagram) $equ\text{-}dgm(\boldsymbol{D})$ for these two functions.

Second. Specify the *standard limit*, *standard limit projection* functions and *standard limiting cone*.

a. The standard limit is the standard equalizer of the equalizer diagram

$\underline{lim}\ D = std\text{-}equ(equ\text{-}dgm(\boldsymbol{D}))$.

b. The standard limit projections are the set function composition of the standard canon and with the projections of the standard product

$\lambda_n : \underline{lim}\ D \to D_n = std\text{-}cnn(equ\text{-}dgm(\boldsymbol{D})) \cdot \pi_n : lim\ D \to \prod_{n \in node(\mathsf{G})} D_n \to D_n$.

c. The standard limiting cone is then constructed out of the standard limit and the standard limit projection functions.

```
(13) (KIF$function tuple)
     (= (KIF$source tuple) set.dgm$diagram)
     (= (KIF$target tuple) set.dgm.tpl$tuple)
     (forall (?d (set.dgm$diagram ?d))
         (and (= (set.dgm.tpl$index (tuple ?d)) (gph$node (set.dgm$shape ?d)))
              (= (tuple ?d) (set.dgm$tuple ?d)))))

(14) (KIF$function target-tuple)
     (= (KIF$source target-tuple) set.dgm$diagram)
     (= (KIF$target target-tuple) set.dgm.tpl$tuple)
     (forall (?d (set.dgm$diagram ?d))
         (and (= (set.dgm.tpl$index (target-tuple ?d)) (gph$edge (set.dgm$shape ?d)))
              (= (target-tuple ?d)
                 (SET.FTN$composition
                     [(gph$target (set.dgm$shape ?d)) (set.dgm$tuple ?d)])))))

(15) (KIF$function target-cone)
     (= (KIF$source target-cone) set.dgm$diagram)
     (= (KIF$target target-cone) set.lim.prd$cone)
     (forall (?d (set.dgm$diagram ?d))
         (and (= (set.lim.prd$cone-tuple (target-cone ?d))
                 (target-tuple ?d))
              (= (set.lim.prd$vertex (target-cone ?d))
                 (set.lim.prd$standard-product (tuple ?d)))
              (forall (?e ((gph$edge (set.dgm$shape ?d)) ?e))
                  (= ((set.lim.prd$component (target-cone ?d)) ?e)
                     ((set.lim.prd$standard-projection (tuple ?d))
                         ((gph$target (set.dgm$shape ?d)) ?e)))))))

(16) (KIF$function target-function)
     (= (KIF$source target-function) set.dgm$diagram)
     (= (KIF$target target-function) set.mor$morphism)
     (forall (?d (set.dgm$diagram ?d))
         (= (target-function ?d)
            (set.lim.prd$mediator (target-cone ?d)))))

(17) (KIF$function source-image-cone)
     (= (KIF$source source-image-cone) set.dgm$diagram)
     (= (KIF$target source-image-cone) set.lim.prd$cone)
     (forall (?d (set.dgm$diagram ?d))
         (and (= (set.lim.prd$cone-tuple (source-image-cone ?d))
                 (target-tuple ?d))
              (= (set.lim.prd$vertex (source-image-cone ?d))
                 (set.lim.prd$standard-product (tuple ?d)))
              (forall (?e ((gph$edge (set.dgm$shape ?d)) ?e))
                  (= ((set.lim.prd$component (source-image-cone ?d)) ?e)
                     (set.ftn$composition
```

```
                        [((set.lim.prd$standard-projection (tuple ?d))
                             ((gph$source (set.dgm$shape ?d)) ?e))
                         ((set.dgm$function ?d) ?e)])))))

(18) (KIF$function source-image-function)
     (= (KIF$source source-image-function) set.dgm$diagram)
     (= (KIF$target source-image-function) set.mor$morphism)
     (forall (?d (set.dgm$diagram ?d))
         (= (source-image-function ?d)
            (set.lim.prd$mediator (source-image-cone ?d))))

(19) (KIF$function parallel-pair)
     (= (KIF$source parallel-pair) set.dgm$diagram)
     (= (KIF$target parallel-pair) set.dgm.ppr$diagram)
     (forall (?d (set.dgm$diagram ?d))
         (and (= (set.dgm.ppr$origin (parallel-pair ?d))
                 (set.lim.prd$standard-product (tuple ?d)))
              (= (set.dgm.ppr$destination (parallel-pair ?d))
                 (set.lim.prd$standard-product (target-tuple ?d)))
              (= (set.dgm.ppr$function1 (parallel-pair ?d))
                 (target-function ?d))
              (= (set.dgm.ppr$function2 (parallel-pair ?d))
                 (source-image-function ?d)))))

(20) (KIF$function standard-limit)
     (= (KIF$source standard-limit) set.dgm$diagram)
     (= (KIF$target standard-limit) set.obj$object)
     (forall (?d (set.dgm$diagram ?d))
         (= (standard-limit ?d)
            (set.lim.equ$standard-equalizer (parallel-pair)))))

(21) (KIF$function standard-projection)
     (= (KIF$source standard-projection) set.dgm$diagram)
     (= (KIF$target standard-projection) SET.FTN$function)
     (forall (?d (set.dgm$diagram ?d))
         (and (= (SET.FTN$source (standard-projection ?d))
                 (gph.obj$node (set.dgm$shape ?d)))
              (= (SET.FTN$target (standard-projection ?d)) set.mor$morphism)
              (= (SET.FTN$composition [(standard-projection ?d) set.mor$source])
                 (SET.FTN$constant [(gph.obj$node (set.dgm$shape ?d)) set.obj$object])
                     (standard-limit ?d)))
              (= (SET.FTN$composition [(standard-projection ?d) set.mor$target])
                 (set.dgm$set ?d))
              (forall (?n ((index (tuple ?d)) ?n))
                  (= ((standard-projection ?d) ?n)
                     (set.ftn$composition
                         [(set.lim.equ$standard-canon (parallel-pair ?d))
                          ((set.lim.prd$standard-projection (tuple ?d)) ?n)])))))

(22) (KIF$function standard-limiting-cone)
     (= (KIF$source standard-limiting-cone) set.dgm$diagram)
     (= (KIF$target standard-limiting-cone) cone)
     (forall (?d (set.dgm$diagram ?d))
         (and (= (cone-diagram (standard-limiting-cone ?d)) ?d)
              (= (vertex (standard-limiting-cone ?d)) ?d) (standard-limit ?d))
              (= (component (standard-limiting-cone ?d)) ?d) (standard-projection ?d))))
```

The *fiber* ***limit-cone***(G) of any graph G is the limiting cone function restricted at source to the diagram fiber and at target to the cone fiber of shape G.

```
(23) (KIF$function limiting-cone-fiber)
     (= (KIF$source limiting-cone-fiber) gph.obj$object)
     (= (KIF$target limiting-cone-fiber) SET.FTN$function)
     (forall (?g (gph.obj$object ?g))
         (and (= (SET.FTN$source (limiting-cone-fiber ?g)) (set.dgm$diagram-fiber ?g))
              (= (SET.FTN$target (limiting-cone-fiber ?g)) (cone-fiber ?g))
              (KIF$restriction (limiting-cone-fiber ?g) limiting-cone)))

(24) (= (SET.FTN$composition [(limiting-cone-fiber ?g) (cone-diagram-fiber ?g)])
        (SET.FTN$identity (set.dgm$diagram-fiber ?g)))
```

```
(25) (KIF$function limit-fiber)
     (= (KIF$source limit-fiber) gph.obj$object)
     (= (KIF$target limit-fiber) SET.FTN$function)
     (forall (?g (gph.obj$object ?g))
         (and (= (SET.FTN$source (limit-fiber ?g)) (set.dgm$diagram-fiber ?g))
              (= (SET.FTN$target (limit-fiber ?g)) set.obj$object)
              (KIF$restriction (limit-fiber ?g) limit)
              (= (limit-fiber ?g)
                 (SET.FTN$composition [(limiting-cone-fiber ?g) (vertex-fiber ?g)])))))

(26) (KIF$function projection-fiber)
     (= (KIF$source projection-fiber) gph.obj$object)
     (= (KIF$target projection-fiber) KIF$function)
     (forall (?g (gph.obj$object ?g))
         (and (= (KIF$source (projection-fiber ?g)) (set.dgm$diagram-fiber ?g))
              (= (KIF$target (projection-fiber ?g)) SET.FTN$function)
              (forall (?d ((set.dgm$diagram-fiber ?g) ?d))
                  (and (= (SET.FTN$source ((projection-fiber ?g) ?d)) (gph.obj$node ?g))
                       (= (SET.FTN$target ((projection-fiber ?g) ?d)) set.mor$morphism)
                       (= ((projection-fiber ?g) ?d) (projection ?d))))))
```

There is a *mediator* set function from the vertex of a cone to the limit of the diagram of the cone. This is the unique set function, which commutes with the component set functions of the cone. We use a KIF definite description to define this. Existence and uniqueness represents the universality of the limit operator.

```
(27) (KIF$function mediator)
     (= (KIF$source mediator) cone)
     (= (KIF$target mediator) set.mor$morphism)
     (forall (?s (cone ?s))
        (= (mediator ?s)
           (the (?m (set.mor$morphism ?m))
                (and (= (set.mor$source ?m) (vertex ?s))
                     (= (set.mor$target ?m) (limit (cone-diagram ?s)))
                     (forall (?n ((gph.obj$node (set.dgm$shape (cone-diagram ?s))) ?n))
                         (= (set.mor$composition [?m ((projection (cone-diagram ?s)) ?n)])
                            ((component ?s) ?n)))))))
```

The *fiber* $comed(G)$ of any graph G is the mediator function restricted at source to the cone fiber.

```
(28) (KIF$function mediator-fiber)
     (= (KIF$source mediator-fiber) gph.obj$object)
     (= (KIF$target mediator-fiber) SET.FTN$function)
     (forall (?g (gph.obj$object ?g))
         (and (= (SET.FTN$source (mediator-fiber ?g)) (cone-fiber ?g))
              (= (SET.FTN$target (mediator-fiber ?g)) set.mor$morphism)
              (= (SET.FTN$composition [(mediator-fiber ?g) set.mor$source])
                 (vertex-fiber ?g))
              (= (SET.FTN$composition [(mediator-fiber ?g) set.mor$target])
                 (SET.FTN$composition [(cone-diagram-fiber ?g) (limit-fiber ?g)]))
              (KIF$restriction (mediator-fiber ?g) mediator)))
```

Limits of Diagram Morphisms

`set.lim.mor`

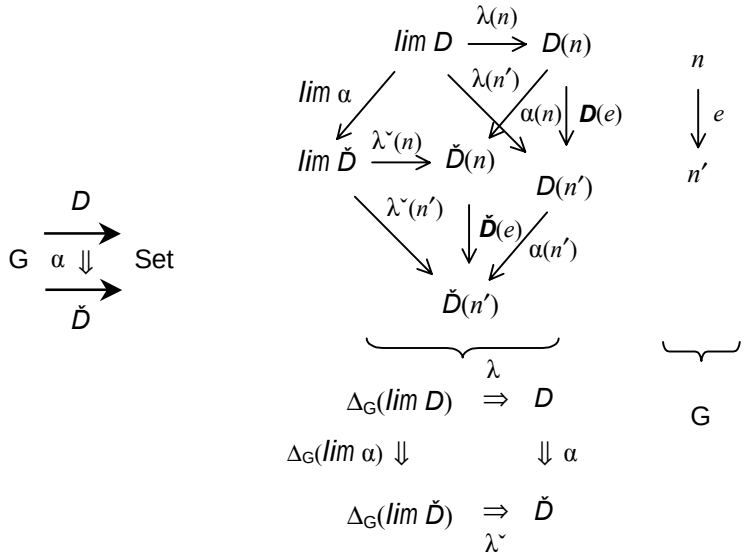


Diagram 13: Diagram Morphisms and Limits

The following three axiom groups can be viewed as axiomatizations of the limit aspect of the exercises 3, 4 and 5 at the end of section V.2 "Limits by Products and Equalizers" in the 1971 edition of Mac Lane's *Categories for the Working Mathematician*.

Limits are defined on diagram morphisms (Diagram 13). Diagram morphisms offer a change of perspective for limits. The limit function maps a set diagram morphism $\boldsymbol{\alpha} : D \Rightarrow \check{D} : \mathsf{G} \to |\mathsf{Set}|$ to a limit set function $lim\ \alpha : lim\ D \to lim\ \check{D}$. As such, it is the morphism function of a limit (quasi)functor $lim : \mathsf{Set}^{\mathsf{G}} \to \mathsf{Set}$ from the functor (quasi)category $\mathsf{Set}^{\mathsf{G}}$ to Set, where the object function is given by the limit function on set diagrams. The limit of a diagram morphism is defined as the mediator of a morphism cone. The limit operator preserves composition and identities.

```
(1) (KIF$function morphism-cone)
    (= (KIF$source morphism-cone) set.dgm.mor$morphism)
    (= (KIF$target morphism-cone) set.lim$cone)
    (forall (?a (set.dgm.mor$morphism ?a))
        (and (= (set.lim$cone-diagram (morphism-cone ?a))
                (set.dgm.mor$target ?a))
             (= (set.lim$vertex (morphism-cone ?a))
                (set.lim$limit (set.dgm.mor$source ?a)))
             (forall (?n ((gph$node (set.dgm.mor$shape ?a)) ?n))
                 (= ((set.lim$component (morphism-cone ?a)) ?n)
                    (SET.FTN$composition
                        [((set.lim$projection (set.dgm.mor$source ?a)) ?n)
                         ((set.dgm.mor$component ?a) ?n)])))))

(2) (KIF$function limit)
    (= (KIF$source limit) set.dgm.mor$morphism)
    (= (KIF$target limit) set.mor$morphism)
    (forall (?a (set.dgm.mor$morphism ?a))
        (and (= (set.mor$source (limit ?a))
                (set.lim$limit (set.dgm.mor$source ?a)))
             (= (set.mor$target (limit ?a))
                (set.lim$limit (set.dgm.mor$target ?a)))
             (= (limit ?a) (set.lim$mediator (morphism-cone ?a))))))

(3) (forall (?a (set.dgm.mor$morphism ?a)
             ?b (set.dgm.mor$morphism ?b)
```

```
           (set.dgm.mor$composable ?a ?b))
         (= (limit (set.dgm.mor$composition [?a ?b]))
            (set.ftn$composition [(limit ?a) (limit ?b)])))

(4) (forall (?d (set.dgm$diagram ?d))
        (= (limit (set.dgm.mor$identity ?d))
           (set.mor$identity (set.lim$limit ?d))))
```

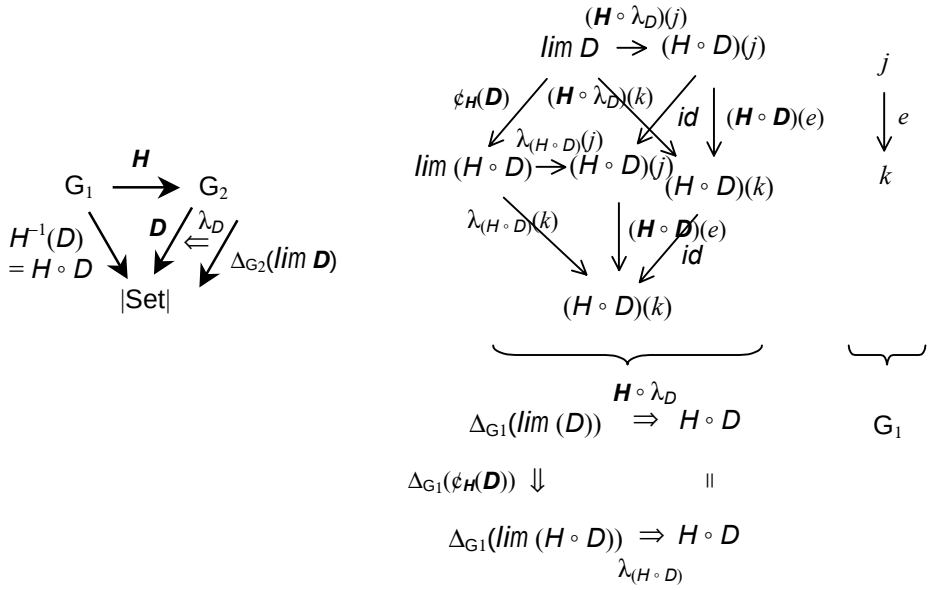


Diagram 14: Limit Connection via Inverse Image

The following operator is needed in order to define limits on lax diagram morphisms (Diagram 14). The limit of a diagram is connected to the limit of its inverse image along a graph morphism. For any graph morphism $H : \mathsf{G}_1 \to \mathsf{G}_2$, the *connection* function maps a diagram $D : \mathsf{G}_2 \to |\mathsf{Set}|$ in the target fiber to a connection set function $\phi_H(D) : \lim (H \circ D) = \lim D \to \lim H^{-1}(D) = \lim (H \circ D)$ from the limit of D to the limit of the inverse image diagram $H^{-1}(D) = H \circ D : \mathsf{G}_1 \to |\mathsf{Set}|$. This function is defined as the mediator of a connection cone.

```
(5) (KIF$function connection-cone)
    (= (KIF$source connection-cone) gph.mor$morphism)
    (= (KIF$target connection-cone) SET.FTN$function)
    (forall (?h (gph.mor$morphism ?h))
        (and (= (SET.FTN$source (connection-cone ?h))
                (set.dgm$diagram-fiber (gph.mor$target ?h)))
             (= (SET.FTN$target (connection-cone ?h))
                (set.lim$cone-fiber (gph.mor$source ?h)))
             (= (SET.FTN$composition
                    [(connection-cone ?h)
                     (set.lim$cone-diagram-fiber (gph.mor$source ?h))])
                ((inverse-image ?h) ?d))
             (= (SET.FTN$composition
                    [(connection-cone ?h)
                     (set.lim$vertex-fiber (gph.mor$source ?h))])
                (set.lim$limit-fiber (gph.mor$target ?h)))
             (forall (?d ((diagram-fiber (gph.mor$target ?h)) ?d))
                 (= ((set.lim$component-fiber (gph.mor$source ?h))
                        ((connection-cone ?h) ?d))
                    (SET.FTN$composition
                        [(gph.mor$node ?h)
                         ((set.lim$projection-fiber (gph.mor$target ?h)) ?d)]))))))

(6) (KIF$function connection)
    (= (KIF$source connection) gph.mor$morphism)
    (= (KIF$target connection) SET.FTN$function)
```

```
(forall (?h (gph.mor$morphism ?h))
    (and (= (SET.FTN$source (connection ?h))
            (set.dgm$diagram-fiber (gph.mor$target ?h)))
         (= (SET.FTN$target (connection ?h)) set.mor$morphism)
         (= (SET.FTN$composition [(connection ?h) set.mor$source])
            (SET.FTN$composition
                [(inverse-image ?h) (set.lim$limit-fiber (gph.mor$source ?h))]))
         (= (SET.FTN$composition [(connection ?h) set.mor$target])
            (set.lim$limit-fiber (gph.mor$target ?h)))
         (= (connection ?h)
            (SET.FTN$composition
                [((connection-cocone ?h)
                 (set.lim$mediator-fiber (gph.mor$source ?h))])))))
```

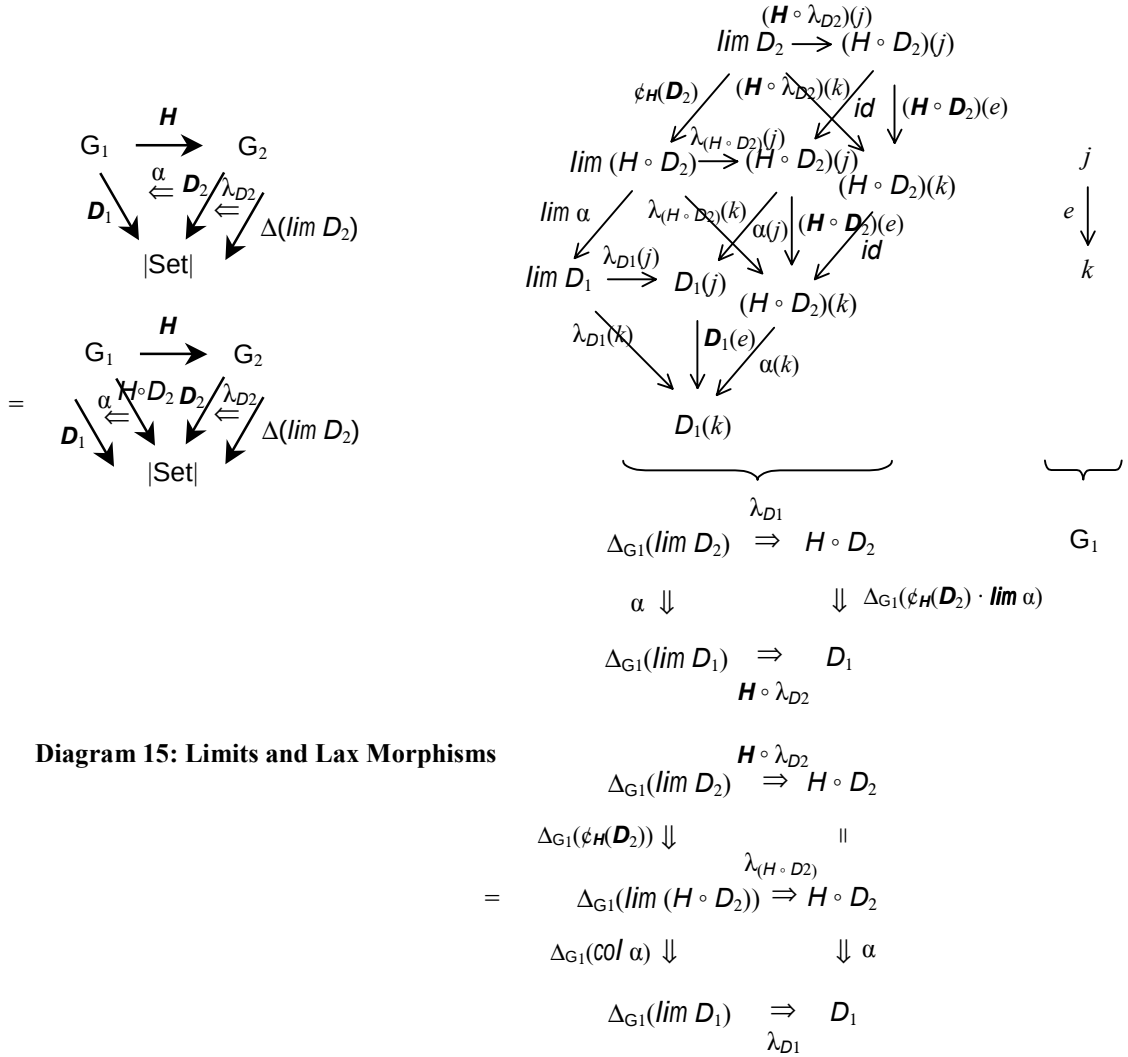


Diagram 15: Limits and Lax Morphisms

We put the previous two ideas together here (Diagram 15). Limits are defined on lax diagram morphisms. Lax diagram morphisms offer a change of perspective for limits. The *lax-limit* function maps a set diagram lax morphism $\mathsf{F} = (H, \alpha) : (\mathsf{G}_1, D_1) \Rightarrow (\mathsf{G}_2, D_2)$ to a lax-limit function $\mathit{lax}\,\mathsf{F} = \phi_H(D_2) \cdot \mathit{lim}\,\alpha : \mathit{lim}\,D \to \mathit{lim}\,\check{D}$. As such, it is the morphism class function of a limit functor $\mathit{lax} : (\mathsf{Cat} \Downarrow \mathsf{Set})^{\mathbf{op}} \to \mathsf{Set}$ from the opposite of the "super-comma" quasi-category with objects being set diagrams and morphisms being lax diagram morphisms. Here, the object function is given by the limit function on set diagrams. The limit of a lax diagram morphism is the composition of the connection of the target diagram $D_2 : \mathsf{G} \to |\mathsf{Set}|$ followed by the limit of the diagram morphism $\boldsymbol{\alpha} : H \circ D_2 \Rightarrow D_1 : \mathsf{G} \to |\mathsf{Set}|$. The lax limit operator preserves composition and identities.

```
(7) (KIF$function lax-limit)
    (= (KIF$source lax-limit) set.dgm.lmor$lax-morphism)
    (= (KIF$target lax-limit) set.mor$morphism)
    (forall (?f (set.dgm.lmor$lax-morphism ?f))
        (and (= (set.mor$source (lax-limit ?f))
                (set.lim$limit (set.dgm.lmor$target ?f)))
             (= (set.mor$target (lax-limit ?f))
                (set.lim$limit (set.dgm.lmor$source ?f)))
             (= (lax-limit ?f)
                (set.mor$composition
                    [((connection (set.dgm.lmor$graph-morphism ?f))
                         (set.dgm.lmor$target ?f))
                     (limit ?a)])))))

(8) (forall (?f (set.dgm.lmor$morphism ?f)
             ?g (set.dgm.lmor$morphism ?g)
            (set.dgm.lmor$composable ?f ?g))
         (= (limit (set.dgm.lmor$composition [?f ?g]))
            (set.ftn$composition [(limit ?g) (limit ?f)]))))

(9) (forall (?d (set.dgm$diagram ?d))
        (= (limit (set.dgm.lmor$identity ?d))
           (set.mor$identity (set.lim$limit ?d))))
```

Terminal Set

There is a special set $0 = \varnothing$ called the *terminal set*. This is the "first" set in the category Set in the following sense: for any set A in Set there is a (*co*) *unique* set function $!_A : 0 \to A$ (the empty function) in Set, which is the unique set function from 0 to A. To express universality we need to know what a cone is for the empty graph shape. This is just an object in Set; that is, a set A. To express universality, we need to axiomatize that $!_A : 0 \to A$ is the unique set function in Set with these source and target sets. Since the empty set is terminal, and all terminal objects (sets) are isomorphic to each other, there is only one terminal object in the category Set, the empty set. We use a definite description to axiomatize (co) uniqueness. A terminal set is the very special case of a colimit under the empty shape graph. A unique function is the very special case of a comediator under the empty shape graph. We connect this specific terminology to the generic fiber terminology for set limits.

```
(10) (set.obj$object terminal)
     (= terminal (set.lim$limit-fiber gph$empty))

(11) (SET.FTN$function unique)
     (= (SET.FTN$source unique set.obj$object)
     (= (SET.FTN$target unique set.mor$morphism)
     (= (SET.FTN$composition [unique set.mor$source])
        (SET.FTN$identity set.obj$object))
     (= (SET.FTN$composition [unique set.mor$target])
        ((SET.FTN$constant [set.obj$object set.obj$object]) terminal))
     (forall (?a (set.obj$object ?a))
         (= (unique ?a)
            (the (?f (set.mor$morphism ?f))
                 (and (= (set.mor$source ?f) ?a)
                      (= (set.mor$target ?f) terminal))))
     (= unique (set.lim$mediator-fiber gph$empty))
```

Products

set.lim.prd

A *product* is a (not necessarily finite) limit in the category Set for a diagram of discrete shape. Such a diagram is called a tuple of sets. Tuples suitable as the underlying diagram for products are axiomatized in the tuple namespace.

The notion of a product *cone* is used to specify and axiomatize products. See the blue part of Diagram 16, where arrows denote set functions. Each product cone is situated under a tuple of sets $\boldsymbol{A} = \{\boldsymbol{A}(n) \mid n \in J\}$ (the green part of Diagram 16). Also, each product cone has a *vertex* set C, and a collection of *component* functions $\tau(n) : C \to \boldsymbol{A}(n)$ indexed by the nodes in the index set J of the tuple. The target sets of the component functions are the component sets of the tuple $\boldsymbol{A}(n)$ and the common source set is the vertex C.

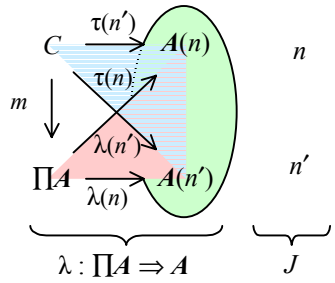


Diagram 16: Product

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

(2) (KIF$function cone-tuple)
    (= (KIF$source cone-tuple) cone)
    (= (KIF$target cone-tuple) set.dgm.tpl$tuple)

(3) (KIF$function vertex)
    (= (KIF$source vertex) cone)
    (= (KIF$target vertex) set.obj$object)

(4) (KIF$function component)
    (= (KIF$source component) cone)
    (= (KIF$target component) SET.FTN$function)
    (forall (?r (cone ?r))
        (and (= (SET.FTN$source (component ?r)) (set.dgm.tpl$index (cone-tuple ?r)))
             (= (SET.FTN$target (component ?r)) set.mor$morphism)
             (= (SET.FTN$composition [(component ?r) set.mor$source])
                ((SET.FTN$constant [(set.dgm.tpl$index (cone-tuple ?r)) set.obj$object])
                    (vertex ?r)))
             (= (SET.FTN$composition [(component ?r) set.mor$target])
                (set.dgm.tpl$set (cone-tuple ?r)))))
```

The *limiting cone* function maps a tuple (discrete diagram) $\boldsymbol{A}$ to its limiting product cone (the red part of Diagram 16). The totality of this function, along with the universality of the mediator set function, implies that a product exists for any tuple of sets. The vertex of the limiting product cone is a specific, but unidentified, *product* set $\prod \boldsymbol{A} = \mathit{prd}(\boldsymbol{A})$. It comes equipped with component *projection* functions $\lambda(n) = \mathit{pr}(\boldsymbol{A})(n) : \prod \boldsymbol{A} = \mathit{prd}(\boldsymbol{A}) \to \boldsymbol{A}(n)$. The product and projections are expressed abstractly by their defining axioms. A limiting product cone is the special case of a general limiting cone under a discrete diagram.

```
(5) (KIF$function limiting-cone)
    (= (KIF$source limiting-cone) set.dgm.tpl$tuple)
    (= (KIF$target limiting-cone) cone)
    (forall (?t (set.dgm.tpl$tuple ?t))
        (= ?t (cone-tuple (limiting-cone ?t))))

(6) (KIF$function limit)
    (KIF$function product)
    (= product limit)
    (= (KIF$source limit) set.dgm.tpl$tuple)
    (= (KIF$target limit) set.obj$object)
    (forall (?t (set.dgm.tpl$tuple ?t))
        (= (limit ?t) (vertex (limiting-cone ?t))))

(7) (KIF$function projection)
    (= (KIF$source projection) set.dgm.tpl$tuple)
    (= (KIF$target projection) SET.FTN$function)
    (forall (?t (set.dgm.tpl$tuple ?t)
        (= (projection ?t) (component (limiting-cone ?t))))
```

For the convenience of the implementer, we also provide a standard product and a standard collection of projections functions that she may choose. The *standard product* or *Cartesian product* of a tuple of sets

$$\textit{std-prd}(\boldsymbol{A}) = \prod_{n \in J} \boldsymbol{A}(n)$$
$$= \{x = (x_n) \mid n \in J, x_n \in \boldsymbol{A}(n)\}$$

provides a concrete coproduct for that tuple. For any tuple $\boldsymbol{A} = \{\boldsymbol{A}(n) \mid n \in J\}$ and any index $n \in J = \textit{ind}(\boldsymbol{A})$ there is a *standard projection* function

$$\textit{std-pro}(\boldsymbol{A})(n) : \textit{std-prd}(\boldsymbol{A}) \to \boldsymbol{A}(n)$$

defined by $\textit{std-pro}(\boldsymbol{A})(n)(x) = x_n$ for all indices $n \in \textit{ind}(\boldsymbol{A})$ and all element $x \in \textit{std-prd}(\boldsymbol{A}) = \prod_{n \in \textit{index}(\boldsymbol{A})} \boldsymbol{A}(n)$. Obviously, the standard projections are surjective. The *standard limiting cone* is then constructed out of the standard product and the standard projection functions. Note how this proceeds bottom-up and in reverse order to the specific components.

```
 (8) (KIF$function standard-limit)
     (KIF$function standard-product)
     (KIF$function cartesian-product)
     (= standard-product standard-limit)
     (= cartesian-product standard-product)
     (= (KIF$source standard-limit) set.dgm.tpl$tuple)
     (= (KIF$target standard-limit) set.obj$object)
     (forall (?a (set.dgm.tpl$tuple ?a))
         (<=> ((standard-limit ?a) ?x)
              (and (set.mor$morphism ?x)
                   (= (set.mor$source ?x) (set.dgm.tpl$index ?a))
                   (= (set.mor$source ?x) (set.mor$tuple-union ?a))
                   (forall (?n ((set.dgm.tpl$index ?a) ?n))
                       (((set.dgm.tpl$set ?a) ?n) (?x ?n)))))))

 (9) (KIF$function standard-projection)
     (= (KIF$source standard-projection) set.dgm.tpl$tuple)
     (= (KIF$target standard-projection) SET.FTN$function)
     (forall (?a (set.dgm.tpl$tuple ?a)
              ?n ((set.dgm.tpl$index ?a) ?n)
              ?x ((standard-limit ?a) ?x))
         (= (((standard-projection ?a) ?n) ?x)
            (?x ?n)))

(10) (SET.FTN$function standard-limiting-cone)
     (= (SET.FTN$source standard-limiting-cone) set.dgm.tpl$diagram)
     (= (SET.FTN$target standard-limiting-cone) cone)
     (= (SET.FTN$composition [standard-limiting-cone cone-tuple])
        (SET.FTN$identity set.dgm.tpl$diagram))
     (= (SET.FTN$composition [standard-limiting-cone vertex]) standard-product)
     (= (SET.FTN$composition [standard-limiting-cone component]) standard-projection)
```

There is a *mediator* function $m : C \to \prod \boldsymbol{A}$ (Diagram 16) from the vertex of a product cone under a tuple of sets to the product of that tuple of sets. This is the unique function that commutes with the component functions of the cone. We have also introduced a "convenience term" tupling. With a tuple parameter, this maps a tuple of set functions, which form a product cone with the tuple, to their mediator (or *tupling*) function. We use a definite description to axiomatize the specific mediator function.

```
(11) (KIF$function mediator)
     (= (KIF$source mediator) cone)
     (= (KIF$target mediator) set.mor$morphism)
     (forall (?r (cone ?r))
        (= (mediator ?r)
           (the (?m (set.mor$morphism ?m))
                (and (= (set.mor$source ?m) (vertex ?r))
                     (= (set.mor$target ?m) (limit (cone-tuple ?r)))
                     (forall (?n ((set.dgm.tpl$index (cone-tuple ?r)) ?n))
                         (= (set.mor$composition
                                [?m ((projection (cone-tuple ?r)) ?n)])
                            ((component ?r) ?n)))))))

(12) (KIF$function tupling-cone)
     (KIF$source tupling-cone) set.dgm.tpl$tuple)
     (KIF$target tupling-cone) SET.FTN$partial-function)
     (forall (?a (set.dgm.tpl$tuple ?a))
         (and (= (SET.FTN$source (tupling-cone ?a))
                 (SET.FTN$power [(set.dgm.tpl$index ?a) set.mor$morphism]))
```

```
            (= (SET.FTN$target (tupling-cone ?a)) cone)
            (forall (?f ((SET.FTN$power [(set.dgm.tpl$index ?a) set.mor$morphism]) ?f))
                (<=> ((SET.FTN$domain (tupling-cone ?a)) ?f)
                     (and (forall (?n ((set.dgm.tpl$index ?a) ?n))
                              (= (set.mor$target (?f ?n)) ((set.dgm.tpl$set ?a) ?n)))
                          (exists (?c (set.obj$object ?c))
                              (forall (?n ((set.dgm.tpl$index ?a) ?n))
                                  (= (set.mor$source (?f ?n)) ?c))))))
            (forall (?f ((SET.FTN$domain (tupling-cone ?a)) ?f))
                (and (= (cone-tuple ((tupling-cone ?a) ?f)) ?a)
                     (forall (?n ((set.dgm.tpl$index ?a) ?n))
                         (= ((component ((tupling-cone ?a) ?f)) ?n) (?f ?n)))))))

(13) (KIF$function tupling)
     (= (KIF$source tupling) set.dgm.tpl$tuple)
     (= (KIF$target tupling) SET.FTN$partial-function)
     (forall (?a (set.dgm.tpl$tuple ?a))
         (and (= (SET.FTN$source (tupling ?a))
                 (SET.FTN$power [(set.dgm.tpl$index ?a) set.mor$morphism]))
              (= (SET.FTN$target (tupling ?a)) set.mor$morphism)
              (= (SET.FTN$domain (tupling ?a))
                 (SET.FTN$domain (tupling-cone ?a)))
              (forall ?f ((SET.FTN$domain (tupling ?a)) ?f))
                  (= ((tupling ?a) ?f)
                     (mediator ((tupling-cone ?a) ?f))))))
```

Binary Products

`set.lim.prd2`

This section is crucial.

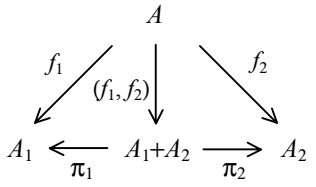


Figure 25: Binary Product Set, Cone and Mediator

A *binary product* (Figure 25) is a finite limit in the category Set for a diagram of shape *two* = • •. Such a diagram (of sets and set functions) is called a *pair* of sets. Given a pair of set A_1 and A_2, a particular binary product is the Cartesian product $A_1 \otimes A_2 = \{(a_1, a_2) \mid a_1 \in A_1, a_2 \in A_2\}$. That is, elements of the Cartesian product are ordered pairs (a_1, a_2) where $a_1 \in A_1$ and $a_2 \in A_2$. The *product projection* set function $\pi_1 = pr(A_1) : A_1 \otimes A_2 \to A_1$ from the Cartesian product $A_1 \otimes A_2$ to the first component set A_1 maps a pairs (a_1, a_2) to the element $a_1 \in A_1$. Since a binary product is a limit, it must satisfy the universality condition for limits. To show universality, let $f_1 : A \to A_1$ and $f_2 : A \to A_2$ be any functions from a common source set A to the components. We need to show that there is a unique set function $(f_1, f_2) : A \to A_1 \otimes A_2$ called the mediator, such that $(f_1, f_2) \cdot \pi_1 = f_1$ and $(f_1, f_2) \cdot \pi_2 = f_2$. By necessity, $(f_1, f_2)(a) = (f_1(a), f_2(a))$ for any element $a \in A$. Let this be the definition of the mediator (f_1, f_2). Hence, the Cartesian product $A_1 \otimes A_2$ is a limit in the category Set. However, as discussed for general limits above, we do not require that the Cartesian product be used for the binary product. Instead, we only require a "moderate expression" for binary products consisting of the description of a map from any pair of sets (diagram) to a limiting cone, with axiomatized assertions that this is a binary product for the pair. Since the Cartesian product is a binary product, and all binary products are isomorphic, the chosen binary product is guaranteed to be isomorphic to the Cartesian product. However, it is not required to be equal to it.

A *binary product cone* is the appropriate cone for a binary product. A product cone (see Figure 25, where arrows denote functions) consists of a pair of set functions called *first* and *second*. These are required to have a common source set called the *vertex* of the cone. Each binary product cone is under a pair of sets. A binary product cone is the very special case of a cone under a pair of sets. We connect this specific terminology to the generic fiber terminology for set limits.

```
(1) (SET$class cone)
    (= cone (set.lim$cone-fiber gph$two)

(2) (SET.FTN$function cone-diagram)
    (= (SET.FTN$source cone-diagram) cone)
    (= (SET.FTN$target cone-diagram) set.dgm.pr$diagram)
    (= cone-diagram (set.lim$cone-diagram-fiber gph$two))

(3) (SET.FTN$function vertex)
    (= (SET.FTN$source vertex) cone)
    (= (SET.FTN$target vertex) set.obj$object)
    (= vertex (set.lim$vertex-fiber gph$two))

(4) (SET.FTN$function first)
    (= (SET.FTN$source first) cone)
    (= (SET.FTN$target first) set.mor$morphism)
    (= (SET.FTN$composition [first set.mor$source])
       (SET.FTN$composition [cone-diagram set.dgm.pr$set1]))
    (= (SET.FTN$composition [first set.mor$target]) vertex)
    (forall (?r (cone ?r))
        (= (first ?r) (((set.lim$component-fiber gph$two) ?r) 1)))

(5) (SET.FTN$function second)
    (= (SET.FTN$source second) cone)
    (= (SET.FTN$target second) set.mor$morphism)
    (= (SET.FTN$composition [second set.mor$source])
       (SET.FTN$composition [cone-diagram set.dgm.pr$set2]))
    (= (SET.FTN$composition [second set.mor$target]) vertex)
    (forall (?r (cone ?r))
        (= (second ?r) (((set.lim$component-fiber gph$two) ?r) 2)))
```

There is a class function called *limiting cone* or *choice* that maps a pair (of sets) to its binary product (limiting binary product cone). The totality of this function, along with the universality of the mediator set

function, implies that a binary product exists for any pair of sets. The vertex of the limiting binary product cone is a specific *binary product* set, which is not necessarily the Cartesian product. It comes equipped with two *projection* functions. The binary product and projections are expressed abstractly by their defining axioms. A binary product limiting cone is the very special case of a limiting cone under a pair of sets. We connect this specific terminology to the generic fiber terminology for set limits.

```
(6) (SET.FTN$function limiting-cone)
    (SET.FTN$function choice)
    (= choice limiting-cone)
    (= (SET.FTN$source limiting-cone) set.dgm.pr$diagram)
    (= (SET.FTN$target limiting-cone) cone)
    (= (SET.FTN$composition [limiting-cone cone-diagram])
       (SET.FTN$identity set.dgm.pr$diagram))
    (= limiting-cone (set.lim$limiting-cone-fiber gph$two))

(7) (SET.FTN$function limit)
    (SET.FTN$function binary-product)
    (= binary-product limit)
    (= (SET.FTN$source limit) set.dgm.pr$diagram)
    (= (SET.FTN$target limit) set.obj$object)
    (= limit (SET.FTN$composition [limiting-cone vertex]))
    (= limit (set.lim$limit-fiber gph$two))

(8) (SET.FTN$function projection1)
    (= (SET.FTN$source projection1) set.dgm.pr$diagram)
    (= (SET.FTN$target projection1) set.mor$morphism)
    (= (SET.FTN$composition [projection1 set.mor$source]) limit)
    (= (SET.FTN$composition [projection1 set.mor$target]) set.dgm.pr$set1)
    (= projection1 (SET.FTN$composition [limiting-cone first]))
    (forall (?d (set.dgm.pr$diagram ?d))
        (= (projection1 ?d) (((set.lim$projection-fiber gph$two) ?d) 1)))

(9) (SET.FTN$function projection2)
    (= (SET.FTN$source projection2) set.dgm.pr$diagram)
    (= (SET.FTN$target projection2) set.mor$morphism)
    (= (SET.FTN$composition [projection2 set.mor$source]) limit)
    (= (SET.FTN$composition [projection2 set.mor$target]) set.dgm.pr$set2)
    (= projection2 (SET.FTN$composition [limiting-cone second]))
    (forall (?d (set.dgm.pr$diagram ?d))
        (= (projection2 ?d) (((set.lim$projection-fiber gph$two) ?d) 2)))
```

For the convenience of the implementer, we also provide a standard binary product and a standard pair of projections functions that she may choose. The *standard product* or *cartesian product* of a pair of sets

$$\mathit{std\text{-}prd}(\boldsymbol{A}) = \boldsymbol{A}(1) \otimes \boldsymbol{A}(2)$$

provides a concrete binary coproduct for that pair. For any pair $\boldsymbol{A}$ there are *standard projection* functions

$$\mathit{std\text{-}pr}(\boldsymbol{A})(1) : \boldsymbol{A}(1) \otimes \boldsymbol{A}(2) \to \boldsymbol{A}(1) \text{ and } \mathit{std\text{-}pr}(\boldsymbol{A})(2) : \boldsymbol{A}(2) \otimes \boldsymbol{A}(2) \to \boldsymbol{A}(2)$$

defined by $\mathit{std\text{-}pr}(\boldsymbol{A})(1)((a_1, a_2)) = a_1$ and $\mathit{std\text{-}inj}(\boldsymbol{A})(2)((a_1, a_2)) = a_2$. Obviously, the standard projections are surjective. The *standard limiting cone* is then constructed out of the standard binary product and the two standard projection functions. Note how this proceeds bottom-up and in reverse order to the specific components.

```
(10) (SET.FTN$function standard-limit)
     (SET.FTN$function standard-binary-product)
     (SET.FTN$function cartesian-binary-product)
     (= standard-binary-product standard-limit)
     (= cartesian-binary-product standard-binary-product)
     (= (SET.FTN$source standard-limit) set.dgm.pr$diagram)
     (= (SET.FTN$target standard-limit) set.obj$object)
     (forall (?p (set.dgm.pr$diagram ?p))
         (<=> ((standard-limit ?p) ?z)
              (and (exists (?x1 ((set.dgm.pr$set1 ?p) ?x1))
                   (exists (?x2 ((set.dgm.pr$set2 ?p) ?x2))
                   (= ?z [?x1 ?x2])))))

(11) (KIF$function standard-projection1)
     (= (KIF$source standard-projection1) set.dgm.pr$diagram)
```

```
     (= (KIF$target standard-projection1) set.mor$morphism)
     (forall (?p (set.dgm.pr$diagram ?p)
              ((standard-limit ?p) ?z))
         (= ((standard-projection1 ?p) ?z) [?z 1]))

(12) (KIF$function standard-projection2)
     (= (KIF$source standard-projection2) set.dgm.pr$diagram)
     (= (KIF$target standard-projection2) set.mor$morphism)
     (forall (?p (set.dgm.pr$diagram ?p)
              ((standard-limit ?p) ?z))
         (= ((standard-projection2 ?p) ?z) [?z 2]))

(13) (SET.FTN$function standard-limiting-cone)
     (= (SET.FTN$source standard-limiting-cone) set.dgm.pr$diagram)
     (= (SET.FTN$target standard-limiting-cone) cone)
     (= (SET.FTN$composition [standard-limiting-cone cone-diagram])
        (SET.FTN$identity set.dgm.pr$diagram))
     (= (SET.FTN$composition [standard-limiting-cone vertex]) standard-binary-product)
     (= (SET.FTN$composition [standard-limiting-cone first]) standard-projection1)
     (= (SET.FTN$composition [standard-limiting-cone second]) standard-projection2)
```

For any binary product cone, there is a *mediator* set function $(f_1, f_2) : A \to A_1 \times A_2$ (see Figure 25, where arrows denote set functions) from the vertex of the cone to the binary product of the underlying diagram (pair of sets). This is the unique function, which commutes with the first f_1 and second f_2 functions. Existence and uniqueness represents the universality of the binary product operator. Universality means that the mediator is the unique function that makes the diagram in Figure 25 commutative. We axiomatize this with a definite description. A binary product mediator is the very special case of a mediator under a pair of sets. We connect this specific terminology to the generic fiber terminology for set limits.

```
(14) (SET.FTN$function mediator)
     (= (SET.FTN$source mediator) cone)
     (= (SET.FTN$target mediator) set.mor$morphism)
     (= (SET.FTN$composition [mediator set.mor$source]) vertex)
     (= (SET.FTN$composition [mediator set.mor$target])
        (SET.FTN$composition [cone-diagram limit]))
     (forall (?r (cone ?r))
         (= (mediator ?r)
            (the (?f (set.mor$morphism ?f))
                (and (= (set.mor$composition [?f (projection1 (cone-diagram ?r))])
                        (first ?r))
                     (= (set.mor$composition [?f (projection2 (cone-diagram ?r))])
                        (second ?r))))))
     (= mediator (set.lim$mediator-fiber gph$two))
```

Subsets and Subobjects

set.lim.sub

This section is crucial.

Subsets are definable on the category Set. Let ***sub*** denote the class of all set subsets. A set *subset* is a pair $\boldsymbol{A} = \langle A, B \rangle = \langle \mathit{set}(\boldsymbol{A}), \mathit{elem}(\boldsymbol{A}) \rangle$ consisting of a set $\mathit{set}(\boldsymbol{A}) = A$ and a subset $\mathit{elem}(\boldsymbol{A}) = B \subseteq A$ of special elements of A. The set A is called the *base set* of $\boldsymbol{A}$ – the set on which $\boldsymbol{A}$ is based. A set subset is determined by the pair consisting of its base set and its subset of elements.

```
(1) (SET$class subset)

(2) (SET.FTN$function set)
    (SET.FTN$function base)
    (= base set)
    (= (SET.FTN$source set) subset)
    (= (SET.FTN$target set) set.obj$object)

(3) (SET.FTN$function element)
    (= (SET.FTN$source element) subset)
    (= (SET.FTN$target element) set.obj$object)
    (forall (?a (subset ?a))
        (set$subset (element ?a) (set ?a)))
```

The class ***sub*** of subsets is ordered. For any two subsets $\boldsymbol{A}_1, \boldsymbol{A}_2 \in \mathit{sub}$, $\boldsymbol{A}_1$ is contained in $\boldsymbol{A}_2$, $\boldsymbol{A}_1 \leq \boldsymbol{A}_2$, when $\mathit{set}(\boldsymbol{A}_1) = \mathit{set}(\boldsymbol{A}_2)$ and $\mathit{elem}(\boldsymbol{A}_1) \subseteq \mathit{elem}(\boldsymbol{A}_2)$.

```
(4) (ORD$partial-order contains)
    (= (ORD$class contains) subset)
    (forall (?a1 (subset ?a1)
             ?a2 (subset ?a2))
        (<=> (contains ?a1 ?a2)
             (and (= (set ?a1) (set ?a2))
                  (set$subset (element ?a1) (element ?a2)))))
```

Any set function $g : B \to A$ defines a subset $\boldsymbol{A}_g = \langle A, \boldsymbol{R}_g \rangle$ called the *range* of g, where the base set is the target set A and the element subset is the range of the function: $\mathit{elem}(\boldsymbol{A}_g) = \boldsymbol{R}_g = \{a \in A \mid a = g(b) \text{ some } b \in B\}$.

```
(5) (SET.FTN$function range)
    (= (SET.FTN$source range) set.mor$morphism)
    (= (SET.FTN$target range) subset)
    (forall (?g (set.mor$morphism ?g))
        (and (= (set (range ?g)) (set.mor$target ?g))
             (= (element (range ?g)) (set.mor$range ?g))))
```

Let $\boldsymbol{A} = \langle A, B \rangle = \langle \mathit{set}(\boldsymbol{A}), \mathit{subset}(\boldsymbol{A}) \rangle$ be a set subset. A set function $g : B \to A$ *respects* $\boldsymbol{A}$ when it factors through $\boldsymbol{A}$; that is, when the range of g is contained in $\boldsymbol{A}$, $\boldsymbol{A}_g \leq \boldsymbol{A}$; that is, when $g(b) \in \mathit{elem}(\boldsymbol{A})$ for any element $b \in B$. By definition, any set function respects its range. Moreover, the range of any set function g is the "smallest" set subset that g respects.

```
(6) (SET.LIM.PBK$opspan matches-opspan)
    (= (SET.LIM.PBK$class1 matches-opspan) set.mor$morphism)
    (= (SET.LIM.PBK$class2 matches-opspan) subset)
    (= (SET.LIM.PBK$opvertex matches-opspan) set.obj$object)
    (= (SET.LIM.PBK$opfirst matches-opspan) set.mor$source)
    (= (SET.LIM.PBK$opsecond matches-opspan) set)

(7) (REL$relation matches)
    (= (REL$class1 matches) set.mor$morphism)
    (= (REL$class2 matches) subset)
    (= (REL$extent matches) (SET.LIM.PBK$pullback matches-opspan))

(8) (REL$relation respects)
    (= (REL$class1 respects) set.mor$morphism)
    (= (REL$class2 respects) subset)
    (REL$subrelation respects matches)
    (forall (?g (set.mor$morphism ?g)
```

```
        ?a (subset ?a) (matches ?g ?a))
    (<=> (respects ?g ?a)
        (contains ?a (range ?g))))
```

Let $\boldsymbol{A} = \langle A, B \rangle = \langle \mathit{set}(\boldsymbol{A}), \mathit{elem}(\boldsymbol{A}) \rangle$ be any set subset.

A *subobject* (Diagram 17) of a set A is a set C plus an injection $m : C \to A$ whose target is A. For any subset $\boldsymbol{A} = \langle A, B \rangle$ with $B \subseteq A$ the inclusion function $\mathit{incl}_B : B \to A$ is the *canonical subobject*. There are many other subobjects, since the source C of the injection m does not need to be a subset of A. However, all subobjects are isomorphic to a subset of A, namely the range of the injection. In particular, given any subset $\boldsymbol{A} = \langle A, B \rangle$, there are many subobjects isomorphic to it, including itself.

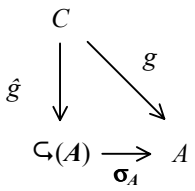


Diagram 17: Canonical Morphism and Universality of the Subobject

The canonical subobject inclusion respects $\boldsymbol{A}$; in fact, the range of the canonical subobject inclusion is the subset of elements B. Also, if any function $g : D \to B$ respects $\boldsymbol{A}$, then $\boldsymbol{A}$ is a superset of the range of g, $\boldsymbol{A}_g \leq \boldsymbol{A}$. In summary, the range of the canonical subobject inclusion incl_B is the largest set subset containing $\boldsymbol{A}$. This expresses a universal property of the canonical subobject.

We abstract this universal property of the canonical subobject as a universality condition, which is stated in the definitions and proposition below. The universality for limits such as equalizers and pullbacks in the category Set are expressed in the IFF in terms of this universality condition. The canonical injection function $[\text{-}]_{\boldsymbol{E}} : A \to A/\equiv_{\boldsymbol{E}}$ satisfies this universality condition. However, this is just one choice of many, all of which are isomorphic to each other.

The *canon*ical injection $\boldsymbol{\sigma}_{\boldsymbol{A}} : \subseteq(\boldsymbol{A}) \to A$ of an subset $\boldsymbol{A} = \langle A, B \rangle = \langle \mathit{set}(\boldsymbol{A}), \mathit{elem}(\boldsymbol{A}) \rangle$ (see Diagram 17, where arrows denote set functions) is a chosen injection that satisfies this universality condition stated in the proposition below. The *subobject* $\subseteq(\boldsymbol{A})$ is the source set of the canonical injection. This canon-subobject pair is called the *canonical subobject*. The totality of the function that chooses the canonical subobject, along with the universality of the mediator set function, axiomatizes the universality condition.

For the convenience of the implementer, for any set subset $\boldsymbol{A} = \langle A, B \rangle = \langle \mathit{set}(\boldsymbol{A}), \mathit{elem}(\boldsymbol{A}) \rangle$ we also provide the *standard subobject*

$\mathit{std\text{-}sub}(\boldsymbol{J}) = B$

and the *standard canon*

$\mathit{std\text{-}cnn}(\boldsymbol{J}) = \mathit{incl}_B : \mathit{std\text{-}sub}(\boldsymbol{J}) = B \to A$

Obviously, the standard canon is an injection.

```
 (9) (SET.FTN$function canon)
     (= (SET.FTN$source canon) subset)
     (= (SET.FTN$target canon) set.mor$injection)
     (= (SET.FTN$composition [canon set.mor$target]) set)

(10) (SET.FTN$function subobject)
     (= (SET.FTN$source subobject) subset)
     (= (SET.FTN$target subobject) set.obj$object)
     (= subobject (SET.FTN$composition [canon set.mor$source]))

(11) (SET.FTN$function standard-subobject)
     (= (SET.FTN$source standard-subobject) subset)
     (= (SET.FTN$target standard-subobject) set.obj$object)
     (= standard-subobject element)

(12) (SET.FTN$function standard-canon)
     (= (SET.FTN$source standard-canon) subset)
     (= (SET.FTN$target standard-canon) set.mor$injection)
     (= (SET.FTN$composition [canon set.mor$target]) set)
     (forall (?a (subset ?a))
         (= (standard-quotient ?a) (set.mor$inclusion [(element ?a) (set ?a)])))
```

Proposition. *Let* $\boldsymbol{A}$ *be any set subset. The canonical injection* $\boldsymbol{\sigma}_{\boldsymbol{A}} : \subsetneq(\boldsymbol{A}) \to A$ *respects* $\boldsymbol{A}$*. Also, any function* $g : C \to A$ *that respects* $\boldsymbol{A}$ *factors uniquely through the canonical subobject; that is, there is a unique mediating function* $\hat{g} : C \to \subsetneq(\boldsymbol{A})$ *such that* $\hat{g} \circ \boldsymbol{\sigma}_{\boldsymbol{A}} = g$.

This can be stated in terms of the order of set subsets: the range of $\boldsymbol{\sigma}_{\boldsymbol{A}}$ is the largest set subset contained in $\boldsymbol{A}$. In more detail, $\boldsymbol{A}$ contains the range of $\boldsymbol{\sigma}_{\boldsymbol{A}}$, $\boldsymbol{A}_{\boldsymbol{\sigma A}} \leq \boldsymbol{A}$; and for any set function $g : C \to A$, if $\boldsymbol{A}$ contains the range of g, $\boldsymbol{A}_{\boldsymbol{g}} \leq \boldsymbol{A}$, then the range of $\boldsymbol{\sigma}_{\boldsymbol{A}}$ contains the range of g, $\boldsymbol{J}_{\boldsymbol{g}} \leq \boldsymbol{J}_{\boldsymbol{\sigma A}}$. Since the canonical injection $\boldsymbol{\sigma}_{\boldsymbol{A}} : \subsetneq(\boldsymbol{A}) \to A$ respects $\boldsymbol{J}$, its unique mediator must be the identity. Based on this proposition, a definite description is used to define a *mediator* function (Diagram 1) that maps a pair $(g, \boldsymbol{A})$ consisting of a set subset and a respectful set function to their mediator $\hat{g}$.

```
(13) (SET.FTN$function mediator)
     (= (SET.FTN$source mediator) (REL$extent respects))
     (= (SET.FTN$target mediator) set.mor$morphism)
     (forall (?g (set.mor$morphism ?g)
              ?a (subset ?a)
              (respects ?g ?a))
         (= (mediator [?g ?a])
            (the (?gt (set.mor$morphism ?gt))
                 (and (= (set.mor$source ?gt) (set.mor$source ?g))
                      (= (set.mor$target ?gt) (subobject ?a))
                      (= (set.mor$composition [?gt (canon ?a)]) ?g))))))
```

From remarks above, the range of the canonical injection $\boldsymbol{\sigma}_{\boldsymbol{A}} : \subsetneq(\boldsymbol{A}) \to A$ is the subset of elements B, the range of the canonical inclusion $incl_B : B \to A$ is also the subset of elements B, and there is a bijection $\subsetneq(\boldsymbol{A}) \cong B$ making $\boldsymbol{\sigma}_{\boldsymbol{A}}$ and $incl_B$ isomorphic.

If the canonical inclusion $incl_B : B \to A$ were chosen as the canon for $\boldsymbol{A}$, the mediator must satisfy $\hat{g}(c) = g(c)$ for all elements $c \in A$. Hence, the respectful constraints on g imply that $\hat{g}$ is well defined. But we reiterate that the chosen subobject $\subsetneq(\boldsymbol{A})$, canon $\boldsymbol{\sigma}_{\boldsymbol{A}} : \subsetneq(\boldsymbol{A}) \to A$ and mediator are not required to be equal to this special case, but only isomorphic to it.

```
(14) (forall (?a (subset ?a))
         (and (set.mor$isomorphic (subobject ?a) (standard-subobject ?a))
              (exists (?h (set.mor$bijection ?h))
                  (= (set.mor$composition [(canon ?a) ?h]) (standard-canon ?a))))))
```

This notion of a set function range gives a canonical approach (Figure 26) for an epi-mono factorization system for set functions. Let μ_g denote the canonical inclusion of the range of g, and let ε_g denote the unique mediating morphism such that $\varepsilon_{\boldsymbol{g}} \circ \mu_{\boldsymbol{g}} = g$. This is an "image factorization" of g. If g respects a set subset $\boldsymbol{A}$, then there is a unique set function $g_A : \boldsymbol{R}_{\boldsymbol{g}} \to \subsetneq(\boldsymbol{A})$ such that $\varepsilon_{\boldsymbol{g}} \circ g_A = \hat{g}$ and $g_A \circ \boldsymbol{\sigma}_A = \mu_{\boldsymbol{g}}$.

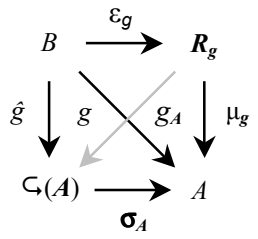


Figure 26: Factorization

Equalizers

`set.lim.equ`

An *equalizer* (Figure 27) is a finite limit in the category Set for a diagram of shape *parallel-pair* = • ⇉ •. Such a diagram is called a *parallel pair* of set functions.

The notion of a *equalizer cone* is used to specify and axiomatize equalizers. Each equalizer cone is situated over an equalizer *diagram* (parallel pair of set functions). And each equalizer cone (see Figure 27, where arrows denote set functions) has a *vertex* set C, and a set function f whose source set is the vertex and whose target set is the origin set of the functions in the overlying cone diagram (parallel-pair). Since Figure 27 is a commutative diagram, the composite function $f \cdot f_1 = f \cdot f_2$ is not needed. An equalizer cone is the very special case of a cone under a parallel pair of functions. We connect this specific terminology to the generic fiber terminology for set limits.

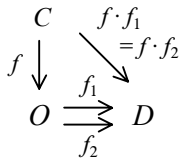


Figure 27: Coequalizer Cocone

```
(1) (SET$class cone)
    (= cone (set.lim$cone-fiber gph$parallel-pair)

(2) (SET.FTN$function cone-diagram)
    (= (SET.FTN$source cone-diagram) cone)
    (= (SET.FTN$target cone-diagram) set.dgm.ppr$diagram)
    (= cone-diagram (set.lim$cone-diagram-fiber gph$parallel-pair))

(3) (SET.FTN$function vertex)
    (= (SET.FTN$source vertex) cone)
    (= (SET.FTN$target vertex) set.obj$object)
    (= vertex (set.lim$vertex-fiber gph$parallel-pair))

(4) (SET.FTN$function function)
    (= (SET.FTN$source function) cone)
    (= (SET.FTN$target function) set.mor$morphism)
    (= (SET.FTN$composition [function set.mor$source]) vertex)
    (= (SET.FTN$composition [function set.mor$target])
       (SET.FTN$composition [cone-diagram set.dgm.ppr$origin]))
    (forall (?r (cone ?r))
        (= (function ?r) (((set.lim$component-fiber gph$parallel-pair) ?r) 1)))

(5) (forall (?r (cone ?r))
       (= (set.mor$composition [(function ?r) (set.dgm.ppr$function1 (cone-diagram ?r))])
          (set.mor$composition [(function ?r) (set.dgm.ppr$function2 (cone-diagram ?r))])))
```

It is important to observe that the set function in a cone respects the subobject of the overlying diagram of the cone. This fact is needed to help define the mediator of a cone.

```
(6) (forall (?r (cone ?r))
       (set.lim.sub$respects (function ?r) (set.dgm.ppr$subobject (cone-diagram ?r))))
```

A *limiting cone* class function maps a parallel pair of set functions to its limiting equalizer cone. See Figure 28, where arrows denote set functions. The totality of this function, along with the universality of the mediator set function, implies that an equalizer exists for any parallel pair of set functions. The vertex of the limiting equalizer cone is a specifically chosen *equalizer* set. It comes equipped with a *canon* set function. The equalizer and canon are expressed both abstractly, and in the last axiom, specifically as the subobject and canon of the set subset of the diagram. That is, the chosen universal objects for set subsets and equalizer diagrams (parallel pairs) are coordinated. An equalizer limiting cone is the very special case of a limiting cone under a parallel pair of functions. We connect this specific terminology to the generic fiber terminology for set limits. The fact that the canon is the "projection" fiber for the graph `parallel-pair`, indicates the abuse of terminology for limit "projections".

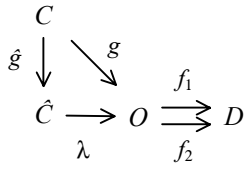


Figure 28: Limiting Cone, Equalizer and Mediator

```
(7) (SET.FTN$function limiting-cone)
    (= (SET.FTN$source limiting-cone) set.dgm.ppr$diagram)
    (= (SET.FTN$target limiting-cone) cone)
    (= (SET.FTN$composition [limiting-cone cone-diagram])
       (SET.FTN$identity set.dgm.ppr$diagram))
    (= limiting-cone (set.lim$limiting-cone-fiber gph$parallel-pair))

(8) (SET.FTN$function limit)
    (SET.FTN$function equalizer)
    (= equalizer limit)
    (= (SET.FTN$source limit) set.dgm.ppr$diagram)
    (= (SET.FTN$target limit) set.obj$object)
    (= limit (SET.FTN$composition [limiting-cone vertex]))
    (= limit (SET.FTN$composition [set.dgm.ppr$subset set.lim.sub$subobject]))
    (= limit (set.lim$limit-fiber gph$parallel-pair))

(9) (SET.FTN$function canon)
    (= (SET.FTN$source canon) set.dgm.ppr$diagram)
    (= (SET.FTN$target canon) set.mor$morphism)
    (= (SET.FTN$composition [canon set.mor$source]) limit])
    (= (SET.FTN$composition [canon set.mor$target]) set.dgm.ppr$origin)
    (= canon (SET.FTN$composition [limiting-cone morphism]))
    (= canon (SET.FTN$composition [set.dgm.ppr$subset set.lim.sub$canon]))
    (forall (?d (set.dgm.pr$diagram ?d))
        (= (canon ?d) (((set.lim$projection-fiber gph$parallel-pair) ?d) 1)))
```

For the convenience of the implementer, for any parallel pair we also provide a *standard equalizer* ***std-eq(p)*** that provides a concrete equalizer for that parallel pair. In addition, there is a *standard canon*

$$\textit{std-cnn}(\boldsymbol{p}) : D \to \textit{std-eq}(\boldsymbol{p}).$$

These are defined in terms of the standard subobject and standard canon of the subset of the pair. The *standard limiting cone* is then constructed out of the standard equalizer and the standard canon functions. Note how this proceeds bottom-up and in reverse order to the specific components.

```
(10) (SET.FTN$function standard-limit)
     (SET.FTN$function standard-equalizer)
     (= standard-equalizer standard-limit)
     (= (SET.FTN$source standard-limit) set.dgm.ppr$diagram)
     (= (SET.FTN$target standard-limit) set.obj$object)
     (= standard-limit
        (SET.FTN$composition [set.dgm.ppr$subset set.lim.sub$standard-subobject]))

(11) (SET.FTN$function standard-canon)
     (= (SET.FTN$source standard-canon) set.dgm.ppr$diagram)
     (= (SET.FTN$target standard-canon) set.mor$morphism)
     (= (SET.FTN$composition [standard-canon set.mor$source]) standard-limit])
     (= (SET.FTN$composition [standard-canon set.mor$target]) set.dgm.ppr$origin)
     (= standard-canon
        (SET.FTN$composition [set.dgm.ppr$subset set.lim.sub$standard-canon]))

(12) (SET.FTN$function standard-limiting-cone)
     (= (SET.FTN$source standard-limiting-cone) set.dgm.ppr$diagram)
     (= (SET.FTN$target standard-limiting-cone) cone)
     (= (SET.FTN$composition [standard-limiting-cone cone-diagram])
        (SET.FTN$identity set.dgm.ppr$diagram))
     (= (SET.FTN$composition [standard-limiting-cone vertex]) standard-equalizer)
     (= (SET.FTN$composition [standard-colimiting-cocone opfirst]) standard-canon)
```

The *mediator* set function, from the vertex of a cone to the equalizer the underlying diagram (parallel pair of set functions), is the unique set function that commutes with cone set functions. See Figure 28, where arrows denote set functions. This is defined using a definite description, which expresses the universal property of the equalizer. It is defined specifically as the mediator of the associated (morphism, set subset) pair. An equalizer mediator is the very special case of a mediator under a parallel pair of functions. We connect this specific terminology to the generic fiber terminology for set limits.

```
(13) (SET.FTN$function mediator)
     (= (SET.FTN$source mediator) cone)
```

```
     (= (SET.FTN$target mediator) set.mor$morphism)
     (= (SET.FTN$composition [mediator set.mor$source]) vertex)
     (= (SET.FTN$composition [mediator set.mor$target])
        (SET.FTN$composition [cone-diagram limit]))
     (forall (?r (cone ?r))
         (= (mediator ?r)
            (the (?m (set.mor$morphism ?m))
                 (= (set.mor$composition [?m (canon (cone-diagram ?r))])
                    (morphism ?r)))))
     (= mediator (set.lim$mediator-fiber gph$parallel-pair))

(14) (forall (?r (cone ?r))
         (= (mediator ?r)
            (set.lim.sub$mediator
                [(morphism ?r) (set.dgm.ppr$subset (cone-diagram ?r))])))
```

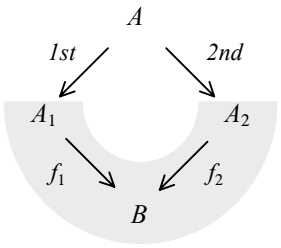
Pullbacks

set.lim.pbk

A *pullback* (Figure 29) is a finite limit in the category Set for a diagram of shape *opspan* = • → • ← •. Such a diagram (of sets and set functions) is called an *opspan*.

Pullback cones are used to specify and axiomatize pullbacks. See Figure 29, where arrows denote set functions. Each pullback cone has an overlying *diagram* (the shaded part of Figure 29), a *vertex* set *A*, and a pair of set functions called *first* and *second*. Their common source set is the vertex and their target sets are the source sets of the functions in the opspan. The first and second functions form a commutative diagram with the opspan. A pullback cone is the very special case of a limiting cone over an opspan. The term 'cone' denotes the *pullback cone* class. The term 'cone-diagram' represents the overlying diagram. A pullback cone is the very special case of a cone under an opspan. We connect this specific terminology to the generic fiber terminology for set limits.

Figure 29: Pullback Cone



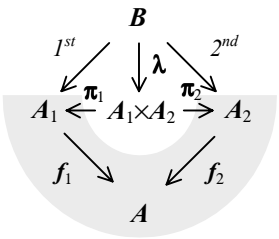

```
(1) (SET$class cone)
    (= cone (set.lim$cone-fiber gph$opspan)

(2) (SET.FTN$function cone-diagram)
    (= (SET.FTN$source cone-diagram) cone)
    (= (SET.FTN$target cone-diagram) set.dgm.ospn$diagram)
    (= cone-diagram (set.lim$cone-diagram-fiber gph$opspan))

(3) (SET.FTN$function vertex)
    (= (SET.FTN$source vertex) cone)
    (= (SET.FTN$target vertex) set.obj$object)
    (= vertex (set.lim$vertex-fiber gph$opspan))

(4) (SET.FTN$function first)
    (= (SET.FTN$source first) cone)
    (= (SET.FTN$target first) set.mor$morphism)
    (= (SET.FTN$composition [first set.mor$source]) vertex)
    (= (SET.FTN$composition [first set.mor$target])
       (SET.FTN$composition [cone-diagram set.dgm.ospn$set1]))
    (forall (?r (cone ?r))
        (= (first ?r) (((set.lim$component-fiber gph$opspan) ?r) 1)))

(5) (SET.FTN$function second)
    (= (SET.FTN$source second) cone)
    (= (SET.FTN$target second) set.mor$morphism)
    (= (SET.FTN$composition [second set.mor$source]) vertex)
    (= (SET.FTN$composition [second set.mor$target])
       (SET.FTN$composition [cone-diagram set.dgm.ospn$set2]))
    (forall (?r (cone ?r))
        (= (second ?r) (((set.lim$component-fiber gph$opspan) ?r) 2)))

(6) (forall (?r (cone ?r))
        (= (set.mor$composition [(first ?r) (set.dgm.ospn$opfirst (cone-diagram ?r))])
           (set.mor$composition [(second ?r) (set.dgm.ospn$opsecond (cone-diagram ?r))])))
```

The *binary-product cone* overlying any cone (pullback diagram) is named.

```
(7) (SET.FTN$function binary-product-cone)
    (= (SET.FTN$source binary-product-cone) cone)
    (= (SET.FTN$target binary-product-cone) set.lim.prd2$cone)
    (= (SET.FTN$composition [binary-product-cone set.lim.prd2$cone-diagram])
       (SET.FTN$composition [cone-diagram set.dgm.ospn$pair]))
    (= (SET.FTN$composition [binary-product-cone set.lim.prd2$vertex]) vertex)
    (= (SET.FTN$composition [binary-product-cone set.lim.prd2$first]) first)
    (= (SET.FTN$composition [binary-product-cone set.lim.prd2$second]) second)
```

The *equalizer cone* function maps a pullback cone of set functions to the associated (`set.lim.equ`) equalizer cone of set functions. See Figure 30, where arrows denote set functions. For this equalizer cone, the overlying equalizer diagram is the equalizer diagram associated with the overlying pullback diagram, the vertex is the vertex of the pullback cone, and the set function is the binary product mediator of the binary product cone (first and second functions with respect to the pair diagram of the cone). This is the first step in the definition of the specific pullback of a pullback cone. The following string of equalities demonstrates that this cone is well defined

Figure 30: Equalizer Cone



$$\lambda \cdot \pi_1 \cdot f_1 = 1^{st} \cdot f_1 = 2^{nd} \cdot f_2 = \lambda \cdot \pi_2 \cdot f_2.$$

```
(8) (SET.FTN$function equalizer-cone)
    (= (SET.FTN$source equalizer-cone) cone)
    (= (SET.FTN$target equalizer-cone) set.lim.equ$cone)
    (= (SET.FTN$composition [equalizer-cone set.lim.equ$cone-diagram])
       (SET.FTN$composition [cone-diagram set.dgm.ospn$parallel-pair]))
    (= (SET.FTN$composition [equalizer-cone set.lim.equ$vertex]) vertex)
    (= (SET.FTN$composition [equalizer-cone set.lim.equ$function])
       (SET.FTN$composition [binary-product-cone set.lim.prd2$mediator]))
```

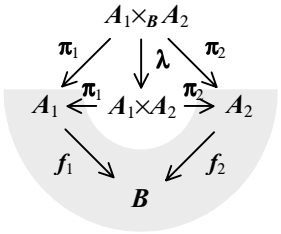


Figure 31: Limiting Cone

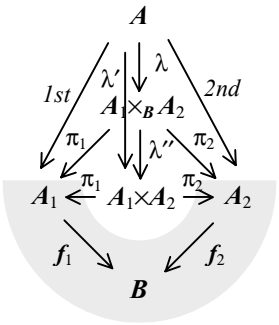


Figure 32: Mediator

The *limiting cone* class function maps an opspan to its limiting pullback cone. See Figure 31, where arrows denote set functions. For convenience of reference, we define three terms that represent the components of this pullback cone. The vertex of the pullback cone is a specific *pullback* set, which comes equipped with two *projection* functions. The last axiom expresses the specificity 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 canon of the equalizer diagram with the product projections of the pair diagram are equal to the pullback projections. These axioms coordinate the choices for products, equalizers and pullbacks. A pullback limiting cone is the very special case of a limiting cone under an opspan of sets and functions. We connect this specific terminology to the generic fiber terminology for set limits.

```
 (9) (SET.FTN$function limiting-cone)
     (= (SET.FTN$source limiting-cone) set.dgm.ospn$diagram)
     (= (SET.FTN$target limiting-cone) cone)
     (= (SET.FTN$composition [limiting-cone cone-diagram])
        (SET.FTN$identity set.dgm.ospn$diagram))
     (= limiting-cone (set.lim$limiting-cone-fiber gph$opspan))

(10) (SET.FTN$function limit)
     (SET.FTN$function pullback)
     (= pullback limit)
     (= (SET.FTN$source limit) set.dgm.ospn$diagram)
     (= (SET.FTN$target limit) set.obj$object)
     (= limit (SET.FTN$composition [limiting-cone vertex]))
     (= limit (set.lim$limit-fiber gph$opspan))

(11) (SET.FTN$function projection1)
```

```
     (= (SET.FTN$source projection1) set.dgm.ospn$diagram)
     (= (SET.FTN$target projection1) set.mor$morphism)
     (= (SET.FTN$composition [projection1 set.mor$source]) limit)
     (= (SET.FTN$composition [projection1 set.mor$target]) set.dgm.ospn$set1)
     (= projection1 (SET.FTN$composition [limiting-cone first])
     (forall (?d (set.dgm.pr$diagram ?d))
         (= (projection1 ?d) (((set.lim$projection-fiber gph$opspan) ?d) 1)))

(12) (SET.FTN$function projection2)
     (= (SET.FTN$source projection2) set.dgm.ospn$diagram)
     (= (SET.FTN$target projection2) set.mor$morphism)
     (= (SET.FTN$composition [projection2 set.mor$source]) set.dgm.ospn$set2)
     (= (SET.FTN$composition [projection2 set.mor$target]) colimit)
     (= projection2 (SET.FTN$composition [limiting-cone second])
     (forall (?d (set.dgm.pr$diagram ?d))
         (= (projection2 ?d) (((set.lim$projection-fiber gph$opspan) ?d) 2)))

(13) (forall (?d (set.dgm.ospn$diagram ?d))
         (and (= (limit ?d)
                 (set.lim.equ$equalizer (set.dgm.ospn$parallel-pair ?d)))
              (= (projection1 ?d)
                 (set.mor$composition
                     [(set.lim.equ$canon (set.dgm.ospn$parallel-pair ?d))
                      (set.lim.prd2$projection1 (set.dgm.ospn$pair ?d))]))
              (= (projection2 ?d)
                 (set.mor$composition
                     [(set.lim.equ$canon (set.dgm.ospn$parallel-pair ?d))
                      (set.lim.prd2$projection2 (set.dgm.ospn$pair ?d))])))))
```

For the convenience of the implementer, for any opspan we also provide a *standard pullback* ***std-pbk***(***s***) = $\boldsymbol{A}(1) \otimes_{\boldsymbol{B}} \boldsymbol{A}(2)$ that provides a concrete pullback for that opspan. In addition, there is a pair of *standard projection* functions

std-pr(***s***)(1) : $\boldsymbol{A}(1) \otimes_{\boldsymbol{B}} \boldsymbol{A}(2) \to \boldsymbol{A}(1)$ and ***std-pr***(***s***)(2) : $\boldsymbol{A}(1) \otimes_{\boldsymbol{B}} \boldsymbol{A}(2) \to \boldsymbol{A}(2)$.

These are defined in terms of the standard product of the pair of the opspan, and the standard equalizer and standard canon of the standard parallel pair of the opspan. The *standard limiting cone* is then constructed out of the standard pullback and the standard projection functions. Note how this proceeds bottom-up and in reverse order to the specific components.

```
(14) (SET.FTN$function standard-limit)
     (SET.FTN$function standard-pullback)
     (= standard-pullback standard-limit)
     (= (SET.FTN$source standard-limit) set.dgm.ospn$diagram)
     (= (SET.FTN$target standard-limit) set.obj$object)

(15) (SET.FTN$function standard-projection1)
     (= (SET.FTN$source standard-projection1) set.dgm.ospn$diagram)
     (= (SET.FTN$target standard-projection1) set.mor$morphism)
     (= (SET.FTN$composition [standard-projection1 set.mor$source]) standard-limit)
     (= (SET.FTN$composition [standard-projection1 set.mor$target]) set.dgm.ospn$set1)

(16) (SET.FTN$function standard-projection2)
     (= (SET.FTN$source standard-projection2) set.dgm.ospn$diagram)
     (= (SET.FTN$target standard-projection2) set.mor$morphism)
     (= (SET.FTN$composition [standard-projection2 set.mor$source]) standard-limit)
     (= (SET.FTN$composition [standard-projection2 set.mor$target]) set.dgm.ospn$set2)

(17) (forall (?d (set.dgm.ospn$diagram ?d))
         (and (= (standard-limit ?d)
                 (set.lim.equ$standard-equalizer (set.dgm.ospn$standard-parallel-pair ?d)))
              (= (standard-projection1 ?d)
                 (set.mor$composition
                     [(set.lim.equ$standard-canon (set.dgm.ospn$standard-parallel-pair ?d))
                      (set.lim.prd2$standard-projection1 (set.dgm.ospn$pair ?d))]))
              (= (standard-projection2 ?d)
                 (set.mor$composition
                     [(set.lim.equ$standard-canon (set.dgm.ospn$standard-parallel-pair ?d))
                      (set.lim.prd2$standard-projection2 (set.dgm.ospn$pair ?d))])))))

(18) (SET.FTN$function standard-limiting-cone)
```

```
     (= (SET.FTN$source standard-limiting-cone) set.dgm.ospn$diagram)
     (= (SET.FTN$target standard-limiting-cone) cone)
     (= (SET.FTN$composition [standard-limiting-cone cone-diagram])
        (SET.FTN$identity set.dgm.ospn$diagram))
     (= (SET.FTN$composition [standard-limiting-cone vertex]) standard-pullback)
     (= (SET.FTN$composition [standard-limiting-cone first]) standard-projection1)
     (= (SET.FTN$composition [standard-limiting-cone second]) standard-projection2)
```

For any pullback cone, the *mediator* set function from the vertex of the cone to the pullback of the overlying diagram of the cone, is the unique set function that commutes with the first and second functions. See Figure 32, where arrows denote set functions. This is defined abstractly by using a definite description, and is defined specifically as the mediator of an equalizer cone. A pullback mediator is the very special case of a mediator under an opspan of sets and functions. We connect this specific terminology to the generic fiber terminology for set limits.

```
(19) (SET.FTN$function mediator)
     (= (SET.FTN$source mediator) cone)
     (= (SET.FTN$target mediator) set.mor$morphism)
     (= (SET.FTN$composition [mediator set.mor$source]) vertex)
     (= (SET.FTN$composition [mediator set.mor$target])
        (SET.FTN$composition [cone-diagram limit]))
     (forall (?r (cone ?r))
         (= (mediator ?r)
            (the (?m (set.mor$morphism ?m))
                 (and (= (composition [?m (projection1 (cone-diagram ?r))])
                         (first ?r))
                      (= (composition [?m (projection2 (cone-diagram ?r))])
                         (second ?r))))))
     (= mediator (set.lim$mediator-fiber gph$opspan))

(20) (= mediator
        (SET.FTN$composition [equalizer-cone set.lim.equ$mediator]))
```

Colimits of Diagrams

`set.col`

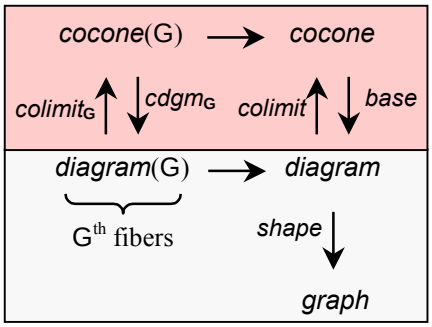


Diagram 18: Diagrams, Cocones and Fibers

Here we present axioms that make Set, the category of sets and set functions, cocomplete. We assert the existence of initial sets, coproducts of collections of sets, coequalizers of parallel pairs of set functions, and pushouts of set spans. All are defined to be specific sets; that is, sets specified by the implementer. In order to prevent clash of common terminology, the axiomatization of binary coproducts, coequalizers and pushouts are put into sub-namespaces. This common terminology has been abstracted into a general formulation of colimits. The *diagrams* and *colimits* in the sub-namespaces are denoted by both generic and specific terminology. A *colimit* is the opvertex of a colimiting cocone of a (base) diagram of a certain shape. The general formulation (Diagram 18) presented first, is parameterized by the shape (graph) of the underlying base diagram for a colimit. The shape of the underlying base diagram uses terminology from the graph namespace of the IFF Lower Core Ontology (IFF-LCO). The colimits of a particular shape diagram are related to the general formulation by an axiomatization for colimit fibers. Many of the other cocomplete categories axiomatized in the IFF lower metalevel make use of Set colimits via there the component functors and natural transformations that describe their objects and morphisms.

Colimits of sets are the most basic. They underlie the colimit construction for any cocomplete category that is axiomatized in the IFF lower metalevel. In particular, they are used in the IFF Ontology (meta) Ontology (IFF-OO) for constructing the *type pole* of object-level ontologies. As demonstrated below, since Set has all sums and coequalizers, it has all colimits – it is cocomplete. There are three expressions for cocompleteness that range along a spectrum of definiteness:

- (weak expression) an axiomatized assertion of existence,
- (moderate expression) the description of a map from any diagram to a colimiting cocone, with axiomatized assertions that this is a colimit for the diagram, and
- (strong expression) an explicit description with axioms of one particular colimit.

The strong expression is most concrete, but perhaps too concrete in that it does not allow choices for implementers. In this namespace and throughout the IFF in general, we describe colimits using the moderate expression; That is, not only do we provide an axiomatized assertion of the existence of colimits, but we also claim, and require in any implementation, the existence of a map from any diagram to a (canonical) colimit. Therefore, while we require a canonical colimit to be described, we leave it up to the implementer how to make that canonical description. Of course, for any particular diagram all such colimits must be unique up to isomorphism.

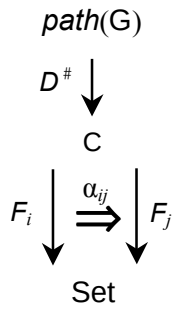
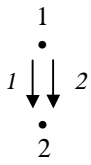
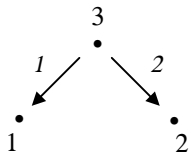


Figure 33: Component Diagrams and Diagram Morphisms

For any cocomplete category C axiomatized in a namespace of the IFF lower metalevel and for any diagram D in C of (small) shape diagram G, it is often the case that the colimiting cocone of D is described implicitly in terms of the colimits of certain diagrams in Set. An illustrative example is the category C = Hypergraph. Figure 33 abstractly describes this situation. Here we assume that objects and morphisms of C are axiomatized (usually uniquely) in terms of certain component structures describe abstractly by component functors (…, F_i, …, F_j, etc.) and natural transformations (…, α_{ij}, etc.) with target category Set. Since Set is cocomplete as axiomatized in this namespace, there are maps from the diagrams, $D^{\#} \circ F_i$, …, $D^{\#} \circ F_j$, etc., to Set colimiting cocones, and there are maps from the diagram morphisms, …, $D^{\#} \circ \alpha_{ij}$, etc., to Set colimiting cocone morphisms. These maps, to Set colimiting cocones and colimiting cocone morphisms, are then used to describe the appropriate map from D to a colimiting cocone in C.

Table 6: Shapes for Diagrams

			
empty (intitial)	*two* (binary coproduct)	*parallel pair* (coequalizer)	*span* (pushout)

In the sections below, we axiomatize colimits for finite diagrams $D : \mathsf{G} \to |\mathsf{Set}|$ of the following finite shape graphs G (Table 2): (i) *empty*, (ii) *two*, (iii) *parallel-pair* and (iv) *span*. These colimits are called (i) the initial set, (ii) a binary coproduct (sum), (iii) a coequalizer and (iv) a pushout. The sections for binary coproducts and endorelations are crucial, since coequalizers, pushouts and other colimits depend upon them.

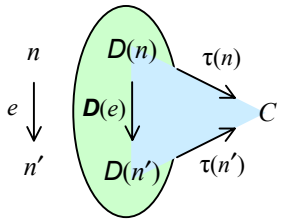


Diagram 19: Cocone

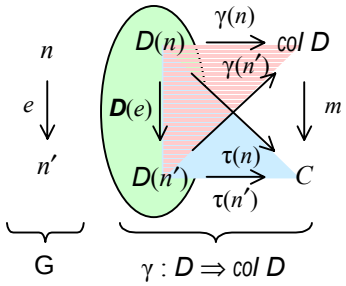


Diagram 20: Colimiting Cocone

A Set-valued *cocone* (Diagram 19) consists of an underlying Set-valued *diagram*, an *opvertex* set, and a collection of *component* set functions indexed by the nodes in the shape of the diagram. A cocone is situated over its diagram. The (external) component set functions of the cocone form commutative diagrams with the (internal) set functions of the underlying diagram.

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

(2) (KIF$function cocone-diagram)
    (= (KIF$source cocone-diagram) cocone)
    (= (KIF$target cocone-diagram) set.dgm$diagram)

(3) (KIF$function opvertex)
    (= (KIF$source opvertex) cocone)
    (= (KIF$target opvertex) set.obj$object)

(4) (KIF$function component)
    (= (KIF$source component) cocone)
    (= (KIF$target component) SET.FTN$function)
    (forall (?s (cocone ?s))
        (and (= (SET.FTN$source (component ?s))
                (gph.obj$node (set.dgm$shape (cocone-diagram ?s))))
             (= (SET.FTN$target (component ?s)) set.mor$morphism)
             (= (SET.FTN$composition [(component ?s) set.mor$source])
                (set.dgm$set (cocone-diagram ?s)))
             (= (SET.FTN$composition [(component ?s) set.mor$target])
                (SET.FTN$constant
                    [(gph.obj$node (set.dgm$shape (cocone-diagram ?s))) set.obj$object]
                        (opvertex ?s)))
             (forall (?e ((gph.obj$edge (set.dgm$shape (cocone-diagram ?s))) ?e))
                 (= (set.mor$composition
                        [((set.dgm$function (cocone-diagram ?s)) ?e)
                         ((component ?s) ((gph.obj$target (set.dgm$shape (cocone-diagram ?s))) ?e))])
                    ((component ?s) ((gph.obj$source (set.dgm$shape (cocone-diagram ?s))) ?e))))))
```

The *fiber* ***cocone***(G) for a (small) shape graph G is the class of all cocones whose underlying diagram has shape G.

```
(5) (KIF$function diagram-shape)
    (= (KIF$source diagram-shape) cocone)
    (= (KIF$target diagram-shape) gph.obj$object)
    (forall (?s (cocone ?s))
        (= (diagram-shape ?s) (set.dgm$shape (cocone-diagram ?s))))

(6) (KIF$function cocone-fiber)
    (= (KIF$source cocone-fiber) gph.obj$object))
    (= (KIF$target cocone-fiber) SET$class)
    (= cocone-fiber (KIF$fiber diagram-shape))

(7) (KIF$function cocone-tuple-fiber)
    (= (KIF$source cocone-tuple-fiber) gph.obj$object)
    (= (KIF$target cocone-tuple-fiber) SET.FTN$function)
    (forall (?g (gph.obj$object ?g))
        (and (= (SET.FTN$source (cocone-tuple-fiber ?g)) (cocone-fiber ?g))
             (= (SET.FTN$target (cocone-tuple-fiber ?g)) (set.dgm$diagram-fiber ?g))
             (forall (?s ((cocone-fiber ?g) ?s))
                 (= ((cocone-tuple-fiber ?g) ?s) (cocone-diagram ?s)))))

(8) (KIF$function opvertex-fiber)
    (= (KIF$source opvertex-fiber) gph.obj$object)
    (= (KIF$target opvertex-fiber) SET.FTN$function)
    (forall (?g (gph.obj$object ?g))
        (and (= (SET.FTN$source (opvertex-fiber ?g)) (cocone-fiber ?g))
             (= (SET.FTN$target (opvertex-fiber ?g)) set.obj$object)
             (forall (?s ((cocone-fiber ?g) ?s))
                 (= ((opvertex-fiber ?g) ?s) (opvertex ?s)))))

(9) (KIF$function component-fiber)
    (= (KIF$source component-fiber) gph.obj$object)
    (= (KIF$target component-fiber) KIF$function)
    (forall (?g (gph.obj$object ?g))
        (and (= (KIF$source (component-fiber ?g)) (cocone-fiber ?g))
             (= (KIF$target (component-fiber ?g)) SET.FTN$function)
             (forall (?s ((cocone-fiber ?g) ?s))
                 (and (= (SET.FTN$source ((component-fiber ?g) ?s)) (gph.obj$node ?g))
                      (= (SET.FTN$target ((component-fiber ?g) ?s)) set.mor$morphism)
                      (= ((component-fiber ?g) ?s) (component ?s)))))))
```

The *colimiting cocone* KIF function maps a generic diagram of sets and set functions to its colimiting cocone (Diagram 20). The opvertex of the colimiting cocone is a specific *colimit* set $\mathit{col}\, D = \mathit{col}(D)$ represented by the *colimit* KIF function. It comes equipped with specific component *injection* (an abuse of the terminology for set coproducts) set functions $\gamma(n) = \mathit{inj}(D)(n) : D(n) \to \mathit{col}\, D = \mathit{col}(D)$. The latter two terms have been created for convenience of reference. These three <u>choice</u> functions assert that a specific colimiting cocone, colimit and collection of injections exist for any diagram. The requirement here is that the implementer should construct a colimit for every diagram, but it does not require a particular colimit to be used. This allows maximum flexibility for implementation. Of course, any chosen colimit is unique up to isomorphism. The universality of this colimit is expressed by axioms for the comediator function.

```
(10) (KIF$function colimiting-cocone)
     (= (KIF$source colimiting-cocone) set.dgm$diagram)
     (= (KIF$target colimiting-cocone) cocone)
     (forall (?d (set.dgm$diagram ?d))
         (= (cocone-diagram (colimiting-cocone ?d)) ?d))

(11) (KIF$function colimit)
     (= (KIF$source colimit) set.dgm$diagram)
     (= (KIF$target colimit) set.obj$object)
     (forall (?d (set.dgm$diagram ?d))
         (= (colimit ?d) (opvertex (colimiting-cocone ?d))))

(12) (KIF$function injection)
     (= (KIF$source injection) set.dgm$diagram)
     (= (KIF$target injection) SET.FTN$function)
     (forall (?d (set.dgm$diagram ?d))
```

```
(and (= (SET.FTN$source (injection ?d))
        (gph.obj$node (set.dgm$shape ?d)))
     (= (SET.FTN$target (injection ?d)) set.mor$morphism)
     (= (SET.FTN$composition [(injection ?d) set.mor$source]) (set.dgm$set ?d))
     (= (SET.FTN$composition [(injection ?d) set.mor$target])
        (SET.FTN$constant [(gph.obj$node (set.dgm$shape ?d)) set.obj$object])
            (colimit ?d)))
     (= (injection ?d) (component (colimiting-cocone ?d)))))
```

For the convenience of the implementer, we also provide a standard colimit and a standard collection of colimit injection functions that she may choose. Here we closely follow the dual to section V.2 of Mac Lane on "Limits by Products and Equalizers". The construction of the *standard colimit* $\underline{\mathit{col}}\ \mathbf{D}$ and *standard injections* $\gamma_n : \mathbf{D}_n \to \underline{\mathit{col}}\ \mathbf{D}$ for a generic diagram $\mathbf{D}$ of sets and set functions is made in two steps. We need to make use of two coproducts in the standard colimit construction.

- Let the following denote the standard coproduct and coproduct projections

  $$\iota_n : \mathbf{D}(n) \to \textstyle\sum \mathit{tpl}(\mathbf{D}) = \sum_{n \in \mathit{node}(\mathsf{G})} \mathbf{D}_n$$

  for the *tuple* (object function) $\mathit{tpl}(\mathbf{D}) : \mathit{node}(\mathsf{G}) \to \mathit{obj}(\mathsf{Set})$ of $\mathbf{D}$ regarded as a tuple.

- In addition, we will also need to use the coproduct and coproduct projections

  $$\iota_e : \mathbf{D}(\mathit{src}(e)) \to \textstyle\sum_{e \in \mathit{edge}(\mathsf{G})} \mathbf{D}_{\mathit{src}(e)} = \sum(\mathit{src}(\mathsf{G}) \cdot \mathit{tpl}(\mathbf{D}))$$

  for the *source tuple* $\mathit{src}(\mathsf{G}) \cdot \mathit{tpl}(\mathbf{D}) : \mathit{edge}(\mathsf{G}) \to \mathit{node}(\mathsf{G}) \to \mathit{obj}(\mathsf{Set})$.

The standard colimit and standard colimit injection functions form a coproduct cocone over the tuple (discrete diagram) $\mathit{tpl}(\mathbf{D})$. Hence, there is a coproduct comediator function $e : \sum_{n \in \mathit{node}(\mathsf{G})} \mathbf{D}_n \to \underline{\mathit{col}}\ \mathbf{D}$ satisfying the constraints $\iota_n \cdot e = \gamma_n$ for all $n \in \mathit{node}(\mathsf{G})$. That is, for any $n \in \mathit{node}(\mathsf{G})$ and any element $x \in \mathbf{D}_n$ of the component set, the element $\gamma_n(x) = e(\iota_n(x)) = e((n, x))$ is an element of the colimit set $\underline{\mathit{col}}\ \mathbf{D}$, where ι_n is the standard coproduct injection. For any edge $e : j \to k$ in the shape graph G of $\mathbf{D}$ and for each element of $x \in \mathbf{D}_j$, the γ-cocone constraints require the identification $\gamma_j(x) = \gamma_k(\mathbf{D}(e)(x))$. For $j = \mathit{src}(e)$ and $k = \mathit{tgt}(e)$, this amounts to requiring that the two coproduct injections $\iota_{\mathit{src}(e)}(x) = (j, x)$ and $\iota_{\mathit{tgt}(e)}(\mathbf{D}(e)(x)) = (k, \mathbf{D}(e)(x))$ be an ordered pair in the equivalence closure of a set endorelation for the standard coequalizer of the parallel pair (Diagram 21) of two set functions

$\iota_{\mathit{src}(e)} : \mathbf{D}_{\mathit{src}(e)} \to \sum_{n \in \mathit{node}(\mathsf{G})} \mathbf{D}_n$ and
$\mathbf{D}_e \cdot \iota_{\mathit{tgt}(e)} : \mathbf{D}_{\mathit{src}(e)} \to \mathbf{D}_{\mathit{tgt}(e)} \to \sum_{n \in \mathit{node}(\mathsf{G})} \mathbf{D}_n$.

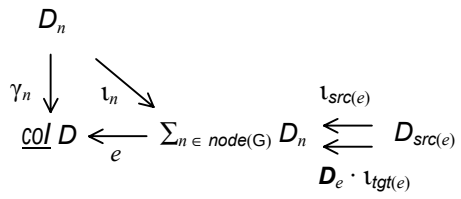


Diagram 21: Parallel Pair for edge e

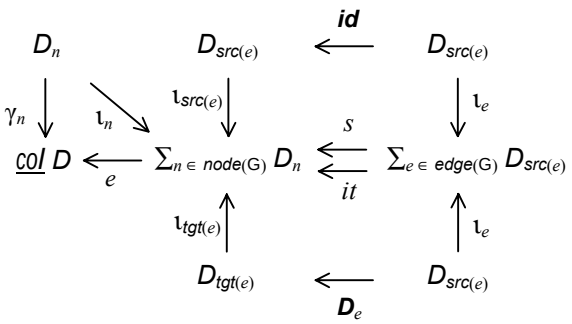


Diagram 22: Colimits in terms of Coequalizers and Coproducts

In the standard colimit construction (Diagram 22) we collect together all of the source domains $D_{src(e)}$ into the coproduct $\prod_{e \in edge(G)} D_{src(e)}$. The following are the steps for computing the standard colimit and its injections in the form of specification statements.

First. Specify the *coequalizer diagram* (parallel pair of set functions).

a. Specify the *source* function

$s : \sum_{e \in edge(G)} D_{tgt(e)} = \sum(tgt(G) \cdot tpl(D)) \to \sum_{n \in node(G)} D_n = \sum tpl(D)$

as the coproduct comediator (or tupling) for the *source cocone*, whose components are the collection of set functions

$\{\iota_{src(e)} : D_{src(e)} \to \sum_{n \in node(G)} D_n = \sum tpl(D) \mid e \in edge(G)\}$.

This is illustrated by the upper square in Diagram 22.

b. Specify the *image target* function

$it : \sum_{e \in edge(G)} D_{tgt(e)} = \sum(tgt(G) \cdot tpl(D)) \to \sum_{n \in node(G)} D_n = \sum tpl(D)$

as the coproduct comediator (or tupling) for the for the *image target cocone*, whose components are the collection of set functions

$\{\boldsymbol{D}_e \cdot \iota_{tgt(e)} : D_{src(e)} \to D_{tgt(e)} \to \sum_{n \in node(G)} D_n \mid e \in edge(G)\}$.

This is illustrated by the lower square in Diagram 22.

c. Form the parallel pair (equalizer diagram) $ppr(\boldsymbol{D})$ for these two functions.

Second. Specify the *standard colimit*, *standard colimit injection* functions and *standard colimiting cocone*.

a. The standard colimit is the standard coequalizer of the coequalizer diagram

$\underline{col}\ D = std\text{-}coeq(ppr(\boldsymbol{D}))$.

b. The standard limit injections are the set function composition of the injections of the standard coproduct and with the standard canon

$\gamma_n : D_n \to \underline{col}\ D = \iota_n \cdot std\text{-}cnn(ppr(\boldsymbol{D})) : D_n \to \sum_{n \in node(G)} D_n \to col\ D$.

c. The standard colimiting cocone is then constructed out of the standard colimit and the standard colimit injection functions.

```
(13) (KIF$function tuple)
     (= (KIF$source tuple) set.dgm$diagram)
     (= (KIF$target tuple) set.dgm.tpl$tuple)
     (forall (?d (set.dgm$diagram ?d))
         (and (= (set.dgm.tpl$index (tuple ?d)) (gph$node (set.dgm$shape ?d)))
              (= (tuple ?d) (set.dgm$tuple ?d))))

(14) (KIF$function source-tuple)
     (= (KIF$source source-tuple) set.dgm$diagram)
     (= (KIF$target source-tuple) set.dgm.tpl$tuple)
     (forall (?d (set.dgm$diagram ?d))
         (and (= (set.dgm.tpl$index (source-tuple ?d)) (gph$edge (set.dgm$shape ?d)))
              (= (source-tuple ?d)
                 (SET.FTN$composition
                     [(gph$source (set.dgm$shape ?d)) (set.dgm$tuple ?d)])))))

(15) (KIF$function source-cocone)
```

```
     (= (KIF$source source-cocone) set.dgm$diagram)
     (= (KIF$target source-cocone) set.lim.prd$cone)
     (forall (?d (set.dgm$diagram ?d))
         (and (= (set.col.coprd$cocone-tuple (source-cocone ?d))
                 (source-tuple ?d))
              (= (set.col.coprd$opvertex (source-cocone ?d))
                 (set.col.coprd$standard-coproduct (tuple ?d)))
              (forall (?e ((gph$edge (set.dgm$shape ?d)) ?e))
                  (= ((set.col.coprd$component (source-cocone ?d)) ?e)
                     ((set.col.coprd$standard-injection (tuple ?d))
                         ((gph$source (set.dgm$shape ?d)) ?e))))))

(16) (KIF$function source-function)
     (= (KIF$source source-function) set.dgm$diagram)
     (= (KIF$target source-function) set.mor$morphism)
     (forall (?d (set.dgm$diagram ?d))
         (= (source-function ?d)
            (set.col.coprd$comediator (source-cone ?d))))

(17) (KIF$function image-target-cocone)
     (= (KIF$source image-target-cocone) set.dgm$diagram)
     (= (KIF$target image-target-cocone) set.col.coprd$cocone)
     (forall (?d (set.dgm$diagram ?d))
         (and (= (set.col.coprd$cocone-tuple (image-target-cocone ?d))
                 (source-tuple ?d))
              (= (set.col.coprd$opvertex (image-target-cocone ?d))
                 (set.col.coprd$standard-coproduct (tuple ?d)))
              (forall (?e ((gph$edge (set.dgm$shape ?d)) ?e))
                  (= ((set.col.coprd$component (image-target-cocone ?d)) ?e)
                     (set.ftn$composition
                         [((set.dgm$function ?d) ?e)
                          ((set.col.coprd$standard-projection (tuple ?d))
                              ((gph$target (set.dgm$shape ?d)) ?e))])))))

(18) (KIF$function image-target-function)
     (= (KIF$source image-target-function) set.dgm$diagram)
     (= (KIF$target image-target-function) set.mor$morphism)
     (forall (?d (set.dgm$diagram ?d))
         (= (image-target-function ?d)
            (set.col.coprd$comediator (image-target-cocone ?d))))

(19) (KIF$function parallel-pair)
     (= (KIF$source parallel-pair) set.dgm$diagram)
     (= (KIF$target parallel-pair) set.dgm.ppr$diagram)
     (forall (?d (set.dgm$diagram ?d))
         (and (= (set.dgm.ppr$origin (parallel-pair ?d))
                 (set.lim.prd$standard-product (source-tuple ?d)))
              (= (set.dgm.ppr$destination (parallel-pair ?d))
                 (set.lim.prd$standard-product (tuple ?d)))
              (= (set.dgm.ppr$function1 (parallel-pair ?d))
                 (source-function ?d))
              (= (set.dgm.ppr$function2 (parallel-pair ?d))
                 (image-target-function ?d)))))

(20) (KIF$function standard-colimit)
     (= (KIF$source standard-colimit) set.dgm$diagram)
     (= (KIF$target standard-colimit) set.obj$object)
     (forall (?d (set.dgm$diagram ?d))
         (= (standard-colimit ?d)
            (set.col.coeq$standard-coequalizer (parallel-pair))))

(21) (KIF$function standard-injection)
     (= (KIF$source standard-injection) set.dgm$diagram)
     (= (KIF$target standard-injection) SET.FTN$function)
     (forall (?d (set.dgm$diagram ?d))
         (and (= (SET.FTN$source (standard-injection ?d))
                 (gph.obj$node (set.dgm$shape ?d)))
              (= (SET.FTN$target (standard-injection ?d)) set.mor$morphism)
              (= (SET.FTN$composition [(standard-injection ?d) set.mor$source])
                 (set.dgm$set ?d))
              (= (SET.FTN$composition [(standard-injection ?d) set.mor$target])
```

```
                (SET.FTN$constant [(gph.obj$node (set.dgm$shape ?d)) set.obj$object])
                    (standard-colimit ?d)))
            (forall (?n ((index (tuple ?d)) ?n))
                (= ((standard-injection ?d) ?n)
                   (set.ftn$composition
                       [((set.col.coprd$standard-injection (tuple ?d)) ?n)
                        (set.col.coeq$standard-canon (parallel-pair ?d))]))))))

(22) (KIF$function standard-colimiting-cocone)
     (= (KIF$source standard-colimiting-cocone) set.dgm$diagram)
     (= (KIF$target standard-colimiting-cocone) cocone)
     (forall (?d (set.dgm$diagram ?d))
         (and (= (cocone-diagram (standard-colimiting-cocone ?d)) ?d)
              (= (opvertex (standard-colimiting-cocone ?d)) ?d) (standard-colimit ?d))
              (= (component (standard-colimiting-cocone ?d)) ?d) (standard-injection ?d))))
```

The *fiber* ***colimit-cocone***(G) of any graph G is the colimiting cocone function restricted at source to the diagram fiber and at target to the cocone fiber of shape G.

```
(23) (KIF$function colimiting-cocone-fiber)
     (= (KIF$source colimiting-cocone-fiber) gph.obj$object)
     (= (KIF$target colimiting-cocone-fiber) SET.FTN$function)
     (forall (?g (gph.obj$object ?g))
         (and (= (SET.FTN$source (colimiting-cocone-fiber ?g)) (set.dgm$diagram-fiber ?g))
              (= (SET.FTN$target (colimiting-cocone-fiber ?g)) (cocone-fiber ?g))
              (KIF$restriction (colimiting-cocone-fiber ?g) colimiting-cocone)))
     (= (SET.FTN$composition [(colimiting-cocone-fiber ?g) (cocone-tuple-fiber ?g)])
        (SET.FTN$identity (set.dgm$diagram-fiber ?g)))

(24) (KIF$function colimit-fiber)
     (= (KIF$source colimit-fiber) gph.obj$object)
     (= (KIF$target colimit-fiber) SET.FTN$function)
     (forall (?g (gph.obj$object ?g))
         (and (= (SET.FTN$source (colimit-fiber ?g)) (set.dgm$diagram-fiber ?g))
              (= (SET.FTN$target (colimit-fiber ?g)) set.obj$object)
              (KIF$restriction (colimit-fiber ?g) colimit)
              (= (colimit-fiber ?g)
                 (SET.FTN$composition [(colimiting-cocone-fiber ?g) (opvertex-fiber ?g)])))))

(25) (KIF$function injection-fiber)
     (= (KIF$source injection-fiber) gph.obj$object)
     (= (KIF$target injection-fiber) KIF$function)
     (forall (?g (gph.obj$object ?g))
         (and (= (KIF$source (injection-fiber ?g)) (set.dgm$diagram-fiber ?g))
              (= (KIF$target (injection-fiber ?g)) SET.FTN$function)
              (forall (?d ((set.dgm$diagram-fiber ?g) ?d))
                  (and (= (SET.FTN$source ((injection-fiber ?g) ?d)) (gph.obj$node ?g))
                       (= (SET.FTN$target ((injection-fiber ?g) ?d)) set.mor$morphism)
                       (= ((injection-fiber ?g) ?d) (injection ?d))))))
```

There is a *comediator* set function from the colimit of the diagram of a cocone to the opvertex of the cocone. This is the unique set function, which commutes with the component set functions of the cocone. We use a KIF definite description to define this. Existence and uniqueness represents the universality of the colimit operator.

```
(26) (KIF$function comediator)
     (= (KIF$source comediator) cocone)
     (= (KIF$target comediator) set.mor$morphism)
     (forall (?s (cocone ?s))
        (= (comediator ?s)
           (the (?m (set.mor$morphism ?m))
                (and (= (set.mor$source ?m) (colimit (cocone-diagram ?s)))
                     (= (set.mor$target ?m) (opvertex ?s))
                     (forall (?n ((gph.obj$node (set.dgm$shape (cocone-diagram ?s))) ?n))
                         (= (set.mor$composition [((injection (cocone-diagram ?s)) ?n) ?m])
                            ((component ?s) ?n)))))))
```

The *fiber* ***comed***(G) of any graph G is the comediator function restricted at source to the cocone fiber.

```
(27) (KIF$function comediator-fiber)
     (= (KIF$source comediator-fiber) gph.obj$object)
```

```
(= (KIF$target comediator-fiber) SET.FTN$function)
(forall (?g (gph.obj$object ?g))
    (and (= (SET.FTN$source (comediator-fiber ?g)) (cocone-fiber ?g))
         (= (SET.FTN$target (comediator-fiber ?g)) set.mor$morphism)
         (= (SET.FTN$composition [(comediator-fiber ?g) set.mor$source])
            (SET.FTN$composition [(cocone-tuple-fiber ?g) (colimit-fiber ?g)]))
         (= (SET.FTN$composition [(comediator-fiber ?g) set.mor$target])
            (opvertex-fiber ?g))
         (KIF$restriction (comediator-fiber ?g) comediator)))
```

Colimits of Diagram Morphisms

`set.col.mor`

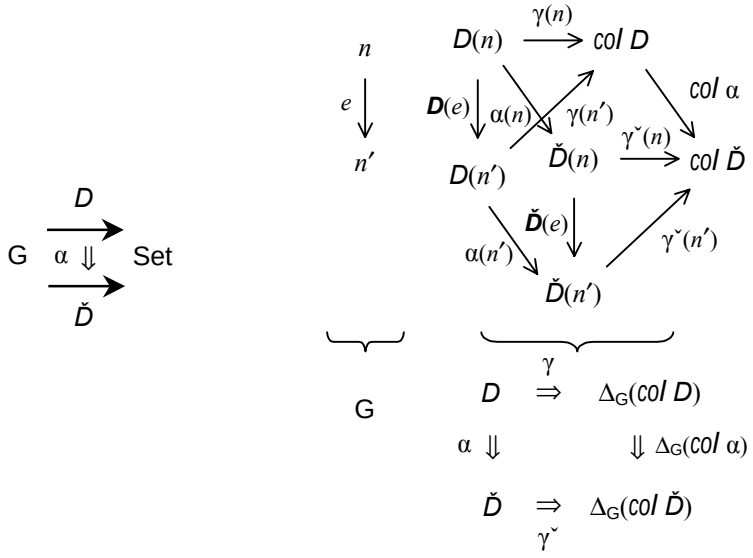


Diagram 23: Diagram Morphisms and Colimits

The following three axiom groups can be viewed as axiomatizations of the colimit aspect of the exercises 3, 4 and 5 at the end of section V.2 "Limits by Products and Equalizers" in the 1971 edition of Mac Lane's *Categories for the Working Mathematician*.

Colimits are defined on diagram morphisms (Diagram 23). Diagram morphisms offer a change of perspective for colimits. The colimit function maps a set diagram morphism $\boldsymbol{\alpha} : D \Rightarrow \check{D} : \mathsf{G} \to |\mathsf{Set}|$ to a colimit set function $\mathit{col}\ \alpha : \mathit{col}\ D \to \mathit{col}\ \check{D}$. As such, it is the morphism function of a colimit (quasi)functor $\mathit{col} : \mathsf{Set}^{\mathsf{G}} \to \mathsf{Set}$ from the functor (quasi)category $\mathsf{Set}^{\mathsf{G}}$ to Set, where the object function is given by the colimit function on set diagrams. The colimit of a diagram morphism is defined as the comediator of a morphism cocone. The colimit operator preserves composition and identities.

```
(1) (KIF$function morphism-cocone)
    (= (KIF$source morphism-cocone) set.dgm.mor$morphism)
    (= (KIF$target morphism-cocone) set.col$cocone)
    (forall (?a (set.dgm.mor$morphism ?a))
        (and (= (set.col$cocone-diagram (morphism-cocone ?a))
                (set.dgm.mor$source ?a))
             (= (set.col$opvertex (morphism-cocone ?a))
                (set.col$colimit (set.dgm.mor$target ?a)))
             (forall (?n ((gph$node (set.dgm.mor$shape ?a)) ?n))
                 (= ((set.col$component (morphism-cocone ?a)) ?n)
                    (SET.FTN$composition
                        [((set.dgm.mor$component ?a) ?n)
                         ((set.col$injection (set.dgm.mor$target ?a)) ?n)])))))

(2) (KIF$function colimit)
    (= (KIF$source colimit) set.dgm.mor$morphism)
    (= (KIF$target colimit) set.mor$morphism)
    (forall (?a (set.dgm.mor$morphism ?a))
        (and (= (set.mor$source (colimit ?a))
                (set.col$colimit (set.dgm.mor$source ?a)))
             (= (set.mor$target (colimit ?a))
                (set.col$colimit (set.dgm.mor$target ?a)))
             (= (colimit ?a) (set.col$comediator (morphism-cocone ?a))))))

(3) (forall (?a (set.dgm.mor$morphism ?a)
             ?b (set.dgm.mor$morphism ?b)
```

```
          (set.dgm.mor$composable ?a ?b))
       (= (colimit (set.dgm.mor$composition [?a ?b]))
          (set.ftn$composition [(colimit ?a) (colimit ?b)])))

(4) (forall (?d (set.dgm$diagram ?d))
       (= (colimit (set.dgm.mor$identity ?d))
          (set.mor$identity (set.col$colimit ?d))))
```

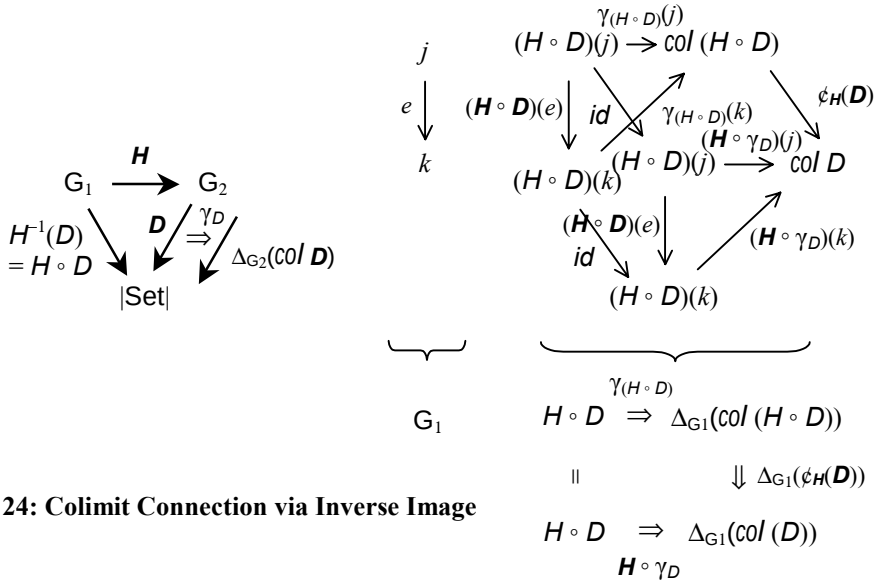


Diagram 24: Colimit Connection via Inverse Image

The following operator is needed in order to define colimits on colax diagram morphisms (Diagram 24). The colimit of a diagram is connected to the colimit of its inverse image along a graph morphism. For any graph morphism $H : \mathsf{G}_1 \to \mathsf{G}_2$, the *connection* function maps a diagram $D : \mathsf{G}_2 \to |\mathsf{Set}|$ in the target fiber to a connection set function $\phi_H(D) : \mathit{col}\ H^{-1}(D) = \mathit{col}\ (H \circ D) \to \mathit{col}\ D$ from the colimit of the inverse image diagram $H^{-1}(D) = H \circ D : \mathsf{G}_1 \to |\mathsf{Set}|$ to the colimit of D. This function is defined as the comediator of a connection cocone.

```
(5) (KIF$function connection-cocone)
    (= (KIF$source connection-cocone) gph.mor$morphism)
    (= (KIF$target connection-cocone) SET.FTN$function)
    (forall (?h (gph.mor$morphism ?h))
        (and (= (SET.FTN$source (connection-cocone ?h))
                (set.dgm$diagram-fiber (gph.mor$target ?h)))
             (= (SET.FTN$target (connection-cocone ?h))
                (set.col$cocone-fiber (gph.mor$source ?h)))
             (= (SET.FTN$composition
                    [(connection-cocone ?h)
                     (set.col$cocone-tuple-fiber (gph.mor$source ?h))])
                (inverse-image ?h))
             (= (SET.FTN$composition
                    [(connection-cocone ?h)
                     (set.col$opvertex-fiber (gph.mor$source ?h))])
                (set.col$colimit-fiber (gph.mor$target ?h)))
             (forall (?d ((diagram-fiber (gph.mor$target ?h)) ?d))
                 (= ((set.col$component-fiber (gph.mor$source ?h))
                        ((connection-cocone ?h) ?d))
                    (SET.FTN$composition
                        [(gph.mor$node ?h)
                         ((set.col$injection-fiber (gph.mor$target ?h)) ?d)]))))))

(6) (KIF$function connection)
    (= (KIF$source connection) gph.mor$morphism)
    (= (KIF$target connection) SET.FTN$function)
    (forall (?h (gph.mor$morphism ?h))
        (and (= (SET.FTN$source (connection ?h))
                (set.dgm$diagram-fiber (gph.mor$target ?h)))
```

```
(= (SET.FTN$target (connection ?h)) set.mor$morphism)
(= (SET.FTN$composition [(connection ?h) set.mor$source])
   (SET.FTN$composition
       [(inverse-image ?h) (set.col$colimit-fiber (gph.mor$source ?h))]))
(= (SET.FTN$composition [(connection ?h) set.mor$target])
   (set.col$colimit-fiber (gph.mor$target ?h)))
(= (connection ?h)
   (SET.FTN$composition
       [((connection-cocone ?h)
        (set.col$comediator-fiber (gph.mor$source ?h))])))
```

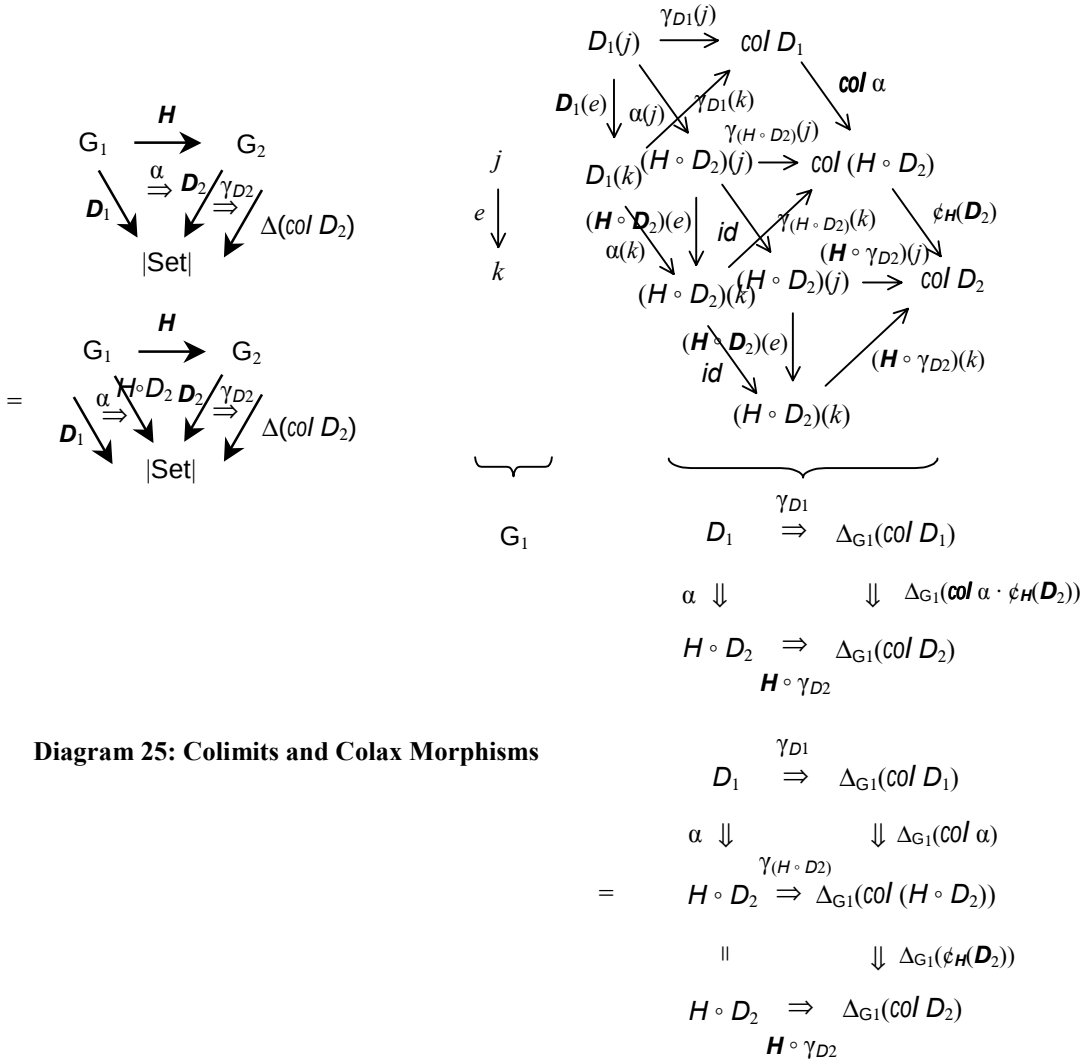


Diagram 25: Colimits and Colax Morphisms

We put the previous two ideas together here (Diagram 25). Colimits are defined on colax diagram morphisms. Colax diagram morphisms offer a change of perspective for colimits. The *colax-colimit* function maps a set diagram colax morphism $\mathsf{F} = (H, \alpha) : (\mathsf{G}_1, D_1) \Rightarrow (\mathsf{G}_2, D_2)$ to a colax-colimit set function $colax\ \mathsf{F} = col\ \alpha \cdot \phi_H(D_2) : col\ D \to col\ \check{D}$. As such, it is the morphism function of a colimit (quasi)functor $colax : \mathsf{Cat} \Uparrow \mathsf{Set} \to \mathsf{Set}$ from the "super-comma" (quasi)category with objects being set diagrams and morphisms being colax diagram morphisms. Here, the object function is given by the colimit function on set diagrams. The colimit of a colax diagram morphism is the composition of the colimit of the diagram morphism $\boldsymbol{\alpha} : D_1 \Rightarrow H \circ D_2 : \mathsf{G} \to |\mathsf{Set}|$ followed by the connection of the target diagram $D_2 : \mathsf{G}_2 \to |\mathsf{Set}|$. The colax colimit operator preserves composition and identities.

```
(7) (KIF$function colax-colimit)
    (= (KIF$source colax-colimit) set.dgm.clmor$colax-morphism)
```

```
    (= (KIF$target colax-colimit) set.mor$morphism)
    (forall (?f (set.dgm.clmor$colax-morphism ?f))
        (and (= (set.mor$source (colax-colimit ?f))
                (set.col$colimit (set.dgm.clmor$source ?f)))
             (= (set.mor$target (colax-colimit ?f))
                (set.col$colimit (set.dgm.clmor$target ?f)))
             (= (colax-colimit ?f)
                (set.mor$composition
                    [(colimit ?a)
                     ((connection (set.dgm.clmor$graph-morphism ?f))
                         (set.dgm.clmor$target ?f))]))))

(8) (forall (?f (set.dgm.clmor$morphism ?f)
             ?g (set.dgm.clmor$morphism ?g)
            (set.dgm.clmor$composable ?f ?g))
         (= (colimit (set.dgm.clmor$composition [?f ?g]))
            (set.ftn$composition [(colimit ?f) (colimit ?g)])))

(9) (forall (?d (set.dgm$diagram ?d))
        (= (colimit (set.dgm.clmor$identity ?d))
           (set.mor$identity (set.col$colimit ?d))))
```

Initial Set

There is a special set $\mathsf{0} = \varnothing$ called the *initial set*. This is the "first" set in the category Set in the following sense: for any set A in Set there is a (*co*) *unique* set function $!_A : \mathsf{0} \to A$ (the empty function) in Set, which is the unique set function from $\mathsf{0}$ to A. To express universality we need to know what a cocone is for the empty graph shape. This is just an object in Set; that is, a set A. To express universality, we need to axiomatize that $!_A : \mathsf{0} \to A$ is the unique set function in Set with these source and target sets. Since the empty set is initial, and all initial objects (sets) are isomorphic to each other, there is only one initial object in the category Set, the empty set. We use a definite description to axiomatize (co) uniqueness. An initial set is the very special case of a colimit over the empty shape graph. A counique function is the very special case of a comediator over the empty shape graph. We connect this specific terminology to the generic fiber terminology for set colimits.

```
(10) (set.obj$object initial)
     (= initial (set.col$colimit-fiber gph$empty))

(11) (SET.FTN$function counique)
     (= (SET.FTN$source counique set.obj$object)
     (= (SET.FTN$target counique set.mor$morphism)
     (= (SET.FTN$composition [counique set.mor$source])
        ((SET.FTN$constant [set.obj$object set.obj$object]) initial))
     (= (SET.FTN$composition [counique set.mor$target])
        (SET.FTN$identity set.obj$object))
     (forall (?a (set.obj$object ?a))
         (= (counique ?a)
            (the (?f (set.mor$morphism ?f))
                 (and (= (set.mor$source ?f) initial)
                      (= (set.mor$target ?f) ?a))))
     (= counique (set.col$comediator-fiber gph$empty))
```

Coproducts of Tuples

`set.col.coprd`

A *coproduct* is a (not necessarily finite) colimit in the category Set for a diagram of (small) discrete shape. Such a diagram is called a tuple of sets. Tuples suitable as the underlying diagram for coproducts are axiomatized in the tuple namespace.

The notion of a coproduct *cocone* is used to specify and axiomatize coproducts. See the blue part of Diagram 26, where arrows denote set functions. Each coproduct cocone is situated over a *tuple* of sets $\boldsymbol{A} = \{\boldsymbol{A}(n) \mid n \in J\}$ (the green part of Diagram 26). Also, each coproduct cocone has an *opvertex* set C, and a collection of *component* functions $\tau(n) : \boldsymbol{A}(n) \to C$ indexed by the nodes in the index set J of the tuple. The source sets of the component functions are the component sets of the tuple $\boldsymbol{A}(n)$ and the common target set is the opvertex C.

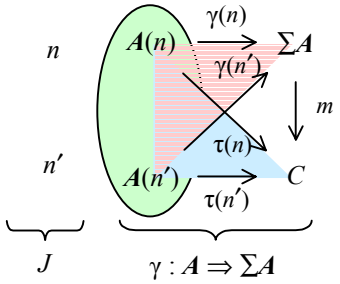


Diagram 26: Coproduct

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

(2) (KIF$function cocone-tuple)
    (= (KIF$source cocone-tuple) cocone)
    (= (KIF$target cocone-tuple) set.dgm.tpl$tuple)

(3) (KIF$function opvertex)
    (= (KIF$source opvertex) cocone)
    (= (KIF$target opvertex) set.obj$object)

(4) (KIF$function component)
    (= (KIF$source component) cocone)
    (= (KIF$target component) SET.FTN$function)
    (forall (?s (cocone ?s))
        (and (= (SET.FTN$source (component ?s)) (set.dgm.tpl$index (cocone-tuple ?s)))
             (= (SET.FTN$target (component ?s)) set.mor$morphism)
             (= (SET.FTN$composition [(component ?s) set.mor$source])
                (set.dgm.tpl$set (cocone-tuple ?s)))
             (= (SET.FTN$composition [(component ?s) set.mor$target])
                ((SET.FTN$constant [(set.dgm.tpl$index (cocone-tuple ?s)) set.obj$object])
                    (opvertex ?s)))))
```

The *fiber* cocone(*J*) for a set *J* is the class of all cocones whose underlying tuple has index set *J*.

```
(5) (KIF$function tuple-index)
    (= (KIF$source tuple-index) cocone)
    (= (KIF$target tuple-index) set.obj$object)
    (forall (?s (cocone ?s))
        (= (tuple-index ?s) (set.dgm.tpl$index (cocone-tuple ?s))))

(6) (KIF$function cocone-fiber)
    (= (KIF$source cocone-fiber) set.obj$object))
    (= (KIF$target cocone-fiber) SET$class)
    (= cocone-fiber (KIF$fiber tuple-index))

(7) (KIF$function cocone-tuple-fiber)
    (= (KIF$source cocone-tuple-fiber) set.obj$object)
    (= (KIF$target cocone-tuple-fiber) SET.FTN$function)
    (forall (?j (set.obj$object ?j))
        (and (= (SET.FTN$source (cocone-tuple-fiber ?j)) (cocone-fiber ?j))
             (= (SET.FTN$target (cocone-tuple-fiber ?j)) (set.dgm.tpl$tuple-fiber ?j))
             (forall (?s ((cocone-fiber ?j) ?s))
                 (= ((cocone-tuple-fiber ?j) ?s) (cocone-diagram ?s)))))

(8) (KIF$function opvertex-fiber)
    (= (KIF$source opvertex-fiber) set.obj$object)
    (= (KIF$target opvertex-fiber) SET.FTN$function)
    (forall (?j (set.obj$object ?j))
        (and (= (SET.FTN$source (opvertex-fiber ?j)) (cocone-fiber ?j))
             (= (SET.FTN$target (opvertex-fiber ?j)) set.obj$object)
             (forall (?s ((cocone-fiber ?j) ?s))
                 (= ((opvertex-fiber ?j) ?s) (opvertex ?s)))))
```

```
(9) (KIF$function component-fiber)
    (= (KIF$source component-fiber) set.obj$object)
    (= (KIF$target component-fiber) KIF$function)
    (forall (?j (set.obj$object ?j))
        (and (= (KIF$source (component-fiber ?j)) (cocone-fiber ?j))
             (= (KIF$target (component-fiber ?j)) SET.FTN$function)
             (forall (?s ((cocone-fiber ?j) ?s))
                 (and (= (SET.FTN$source ((component-fiber ?j) ?s)) ?j)
                      (= (SET.FTN$target ((component-fiber ?j) ?s)) set.mor$morphism)
                      (= ((component-fiber ?j) ?s) (component ?s))))))
```

The *colimiting cocone* function maps a tuple (discrete diagram) $\boldsymbol{A}$ to its colimiting coproduct cocone (the red part of Diagram 26). The totality of this function, along with the universality of the comediator set function, implies that a coproduct exists for any tuple of sets. The opvertex of the colimiting coproduct cocone is a specific, but unidentified, *coproduct* set $\sum \boldsymbol{A} = \mathit{coprd}(\boldsymbol{A})$. It comes equipped with component *injection* functions $\gamma(n) = \mathit{inj}(\boldsymbol{A})(n) : \boldsymbol{A}(n) \to \sum \boldsymbol{A} = \mathit{coprd}(\boldsymbol{A})$. The coproduct and injections are expressed abstractly by their defining axioms. A colimiting coproduct cocone is the special case of a general colimiting cocone over a discrete diagram.

```
(10) (KIF$function colimiting-cocone)
     (= (KIF$source colimiting-cocone) set.dgm.tpl$tuple)
     (= (KIF$target colimiting-cocone) cocone)
     (forall (?a (set.dgm.tpl$tuple ?a))
         (= ?a (cocone-tuple (colimiting-cocone ?a))))

(11) (KIF$function colimit)
     (KIF$function coproduct)
     (= coproduct colimit)
     (= (KIF$source colimit) set.dgm.tpl$tuple)
     (= (KIF$target colimit) set.obj$object)
     (forall (?a (set.dgm.tpl$tuple ?a))
         (= (colimit ?a) (opvertex (colimiting-cocone ?a))))

(12) (KIF$function injection)
     (= (KIF$source injection) set.dgm.tpl$tuple)
     (= (KIF$target injection) SET.FTN$function)
     (forall (?a (set.dgm.tpl$tuple ?a)
         (= (injection ?a) (component (colimiting-cocone ?a))))
```

For the convenience of the implementer, we also provide a standard coproduct and a standard collection of injections functions that she may choose. The *standard coproduct* or *disjoint union* of a tuple of sets

$$\begin{aligned} \mathit{std\text{-}coprd}(\boldsymbol{A}) &= \textstyle\sum_{n \in J} \boldsymbol{A}(n) \\ &= \{(n, x) \mid n \in J, x \in \boldsymbol{A}(n)\} \end{aligned}$$

provides a concrete coproduct for that tuple. For any tuple $\boldsymbol{A} = \{\boldsymbol{A}(n) \mid n \in J\}$ and any index $n \in \mathit{index}(\boldsymbol{A})$ there is a *standard injection* function

$$\mathit{std\text{-}inj}(\boldsymbol{A})(n) : \boldsymbol{A}(n) \to \mathit{std\text{-}coprd}(\boldsymbol{A})$$

defined by $\mathit{std\text{-}inj}(\boldsymbol{A})(n)(x) = (n, x)$ for all indices $n \in \mathit{ind}(\boldsymbol{A})$ and all elements $x \in \boldsymbol{A}(n)$. Obviously, the standard injections are injective. The *standard colimiting cocone* is then constructed out of the standard coproduct and the standard injection functions. Note how this proceeds bottom-up and in reverse order to the specific components.

```
(13) (KIF$function standard-colimit)
     (KIF$function standard-coproduct)
     (KIF$function disjoint-union)
     (= standard-coproduct standard-colimit)
     (= disjoint-union standard-coproduct)
     (= (KIF$source standard-colimit) set.dgm.tpl$tuple)
     (= (KIF$target standard-colimit) set.obj$object)
     (forall (?a (set.dgm.tpl$tuple ?a))
         (<=> ((standard-colimit ?a) ?x)
              (exists (?n ((set.dgm.tpl$index ?a) ?n)
                       ?xn (((set.dgm.tpl$set ?a) ?n) ?xn))
                  (= ?x [?n ?xn])))))

(14) (KIF$function standard-injection)
```

```
     (= (KIF$source standard-injection) set.dgm.tpl$tuple)
     (= (KIF$target standard-injection) SET.FTN$function)
     (forall (?a (set.dgm.tpl$tuple ?a)
              ?n ((set.dgm.tpl$index ?a) ?n)
              ?xn (((set.dgm.tpl$set ?a) ?n) ?xn))
         (= (((standard-injection ?a) ?n) ?xn) [?n ?xn]))

(15) (SET.FTN$function standard-colimiting-cocone)
     (= (SET.FTN$source standard-colimiting-cocone) set.dgm.tpl$diagram)
     (= (SET.FTN$target standard-colimiting-cocone) cocone)
     (= (SET.FTN$composition [standard-colimiting-cocone cocone-tuple])
        (SET.FTN$identity set.dgm.tpl$diagram))
     (= (SET.FTN$composition [standard-colimiting-cocone opvertex]) standard-coproduct)
     (= (SET.FTN$composition [standard-colimiting-cocone component]) standard-injection)
```

The *fiber* colimit-cocone(J) for any set J is the colimiting cocone function restricted at source to the tuple fiber and at target to the cocone fiber of index J.

```
(16) (KIF$function colimiting-cocone-fiber)
     (= (KIF$source colimiting-cocone-fiber) set.obj$object)
     (= (KIF$target colimiting-cocone-fiber) SET.FTN$function)
     (forall (?j (set.obj$object ?j))
         (and (= (SET.FTN$source (colimiting-cocone-fiber ?j)) (set.dgm$tuple-fiber ?j))
              (= (SET.FTN$target (colimiting-cocone-fiber ?j)) (cocone-fiber ?j))
              (KIF$restriction (colimiting-cocone-fiber ?j) colimiting-cocone)))
     (= (SET.FTN$composition [(colimiting-cocone-fiber ?j) (cocone-tuple-fiber ?j)])
        (SET.FTN$identity (set.dgm.tpl$tuple-fiber ?j)))

(17) (KIF$function colimit-fiber)
     (KIF$function coproduct-fiber)
     (= coproduct-fiber colimit-fiber)
     (= (KIF$source colimit-fiber) set.obj$object)
     (= (KIF$target colimit-fiber) SET.FTN$function)
     (forall (?j (set.obj$object ?j))
         (and (= (SET.FTN$source (colimit-fiber ?j)) (set.dgm.tpl$tuple-fiber ?j))
              (= (SET.FTN$target (colimit-fiber ?j)) set.obj$object)
              (KIF$restriction (colimit-fiber ?j) colimit)
              (= (colimit-fiber ?j)
                 (SET.FTN$composition [(colimiting-cocone-fiber ?j) (opvertex-fiber ?j)])))))

(18) (KIF$function injection-fiber)
     (= (KIF$source injection-fiber) set.obj$object)
     (= (KIF$target injection-fiber) KIF$function)
     (forall (?j (set.obj$object ?j))
         (and (= (KIF$source (injection-fiber ?j)) (set.dgm.tpl$tuple-fiber ?j))
              (= (KIF$target (injection-fiber ?j)) SET.FTN$function)
              (forall (?a ((set.dgm.tpl$tuple-fiber ?j) ?a))
                  (and (= (SET.FTN$source ((injection-fiber ?j) ?a)) ?j)
                       (= (SET.FTN$target ((injection-fiber ?j) ?a)) set.mor$morphism)
                       (= ((injection-fiber ?j) ?a) (injection ?a)))))))
```

There is a *comediator* function $m : \sum \boldsymbol{A} \to C$ (Diagram 26) to the opvertex of a coproduct cocone over a tuple of sets from the coproduct of that tuple of sets. This is the unique function that commutes with the component functions of the cocone. We have also introduced a "convenience term" cotupling. With a tuple parameter, this maps a tuple of set functions, which form a coproduct cocone with the tuple, to their comediator (or *cotupling*) function. We use a definite description to axiomatize the specific mediator function.

```
(19) (KIF$function comediator)
     (= (KIF$source comediator) cocone)
     (= (KIF$target comediator) set.mor$morphism)
     (forall (?s (cocone ?s))
        (= (comediator ?s)
           (the (?m (set.mor$morphism ?m))
                (and (= (set.mor$source ?m) (colimit (cocone-tuple ?s)))
                     (= (set.mor$target ?m) (opvertex ?s))
                     (forall (?n ((set.dgm.tpl$index (cocone-tuple ?s)) ?n))
                         (= (set.mor$composition
                                [((injection (cocone-tuple ?s)) ?n) ?m])
                            ((component ?s) ?n)))))))
```

```
(20) (KIF$function cotupling-cocone)
     (KIF$source cotupling-cocone) set.dgm.tpl$tuple)
     (KIF$target cotupling-cocone) SET.FTN$partial-function)
     (forall (?a (set.dgm.tpl$tuple ?a))
         (and (= (SET.FTN$source (cotupling-cocone ?a))
                 (SET.FTN$power [(set.dgm.tpl$index ?a) set.mor$morphism]))
              (= (SET.FTN$target (cotupling-cocone ?a)) cocone)
              (forall (?f ((SET.FTN$power [(set.dgm.tpl$index ?a) set.mor$morphism]) ?f))
                  (<=> ((SET.FTN$domain (cotupling-cocone ?a)) ?f)
                       (and (forall (?n ((set.dgm.tpl$index ?a) ?n))
                                    (= (set.mor$source (?f ?n)) ((set.dgm.tpl$set ?a) ?n)))
                            (exists (?c (set.obj$object ?c))
                                (forall (?n ((set.dgm.tpl$index ?a) ?n))
                                    (= (set.mor$target (?f ?n)) ?c))))))
              (forall (?f ((SET.FTN$domain (cotupling-cocone ?a)) ?f))
                  (and (= (cocone-tuple ((cotupling-cocone ?a) ?f)) ?a)
                       (forall (?n ((set.dgm.tpl$index ?a) ?n))
                           (= ((component ((cotupling-cocone ?a) ?f)) ?n) (?f ?n)))))))

(21) (KIF$function cotupling)
     (= (KIF$source cotupling) set.dgm.tpl$tuple)
     (= (KIF$target cotupling) SET.FTN$partial-function)
     (forall (?a (set.dgm.tpl$tuple ?a))
         (and (= (SET.FTN$source (cotupling ?a))
                 (SET.FTN$power [(set.dgm.tpl$index ?a) set.mor$morphism]))
              (= (SET.FTN$target (cotupling ?a)) set.mor$morphism)
              (= (SET.FTN$domain (cotupling ?a))
                 (SET.FTN$domain (cotupling-cocone ?a)))
              (forall ?f ((SET.FTN$domain (cotupling ?a)) ?f))
                  (= ((cotupling ?a) ?f)
                     (comediator ((cotupling-cocone ?a) ?f))))))
```

There is an *indication* function (special for coproducts) (Diagram 27)

$\mathit{indic}(\boldsymbol{A}) : \mathit{coprd}(\boldsymbol{A}) \to \mathit{ind}(\boldsymbol{A})$

from the coproduct to the tuple index, defined by $\mathit{indic}(\boldsymbol{A})(n, x) = n$. The indication function is the cotupling of the collection of constant index functions

$\{\Delta_{\boldsymbol{A}}(n) : \boldsymbol{A}(n) \to \mathit{ind}(\boldsymbol{A}) \mid n \in \mathit{ind}(\boldsymbol{A})\}$,

where $\Delta_{\boldsymbol{A}}(n)(x_n) = n$ for all $n \in \mathit{ind}(\boldsymbol{A})$ and $x_n \in \boldsymbol{A}(n)$.

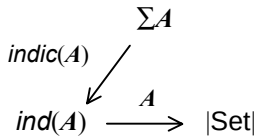


Diagram 27: Colimit and Indication

```
(22) (KIF$function constant-index)
     (= (KIF$source constant-index) set.dgm.tpl$tuple)
     (= (KIF$target constant-index) SET.FTN$function)
     (forall (?a (set.dgm.tpl$tuple ?a))
         (and (= (SET.FTN$source (constant-index ?a)) (set.dgm.tpl$index ?a))
              (= (SET.FTN$target (constant-index ?t)) set.mor$morphism)
              (forall (?n ((set.dgm.tpl$index ?a) ?n))
                  (and (= (set.mor$source ((constant-index ?a) ?n)) ((set.dgm.tpl$set ?a) ?n))
                       (= (set.mor$target ((constant-index ?a) ?n)) (set.dgm.tpl$index ?a))
                       (= ((constant-index ?a) ?n)
                          ((set.mor$constant
                               [((set.dgm.tpl$set ?a) ?n) (set.dgm.tpl$index ?a)]) ?n))))))

(23) (SET.FTN$function indication)
     (= (SET.FTN$source indication) set.dgm.tpl$tuple)
     (= (SET.FTN$target indication) set.mor$morphism)
     (forall (?a (set.dgm.tpl$tuple ?a)
         (and (= (set.mor$source (indication ?a)) (coproduct ?a))
              (= (set.mor$target (indication ?a)) (set.dgm.tpl$index ?a))
              (= (indication ?a) ((cotupling ?a) (constant-index ?a)))))
```

The *fiber* $\mathit{comed}(J)$ for any set J is the comediator function restricted at source to the cocone fiber.

```
(24) (KIF$function comediator-fiber)
     (= (KIF$source comediator-fiber) set.obj$object)
     (= (KIF$target comediator-fiber) SET.FTN$function)
     (forall (?j (set.obj$object ?j))
         (and (= (SET.FTN$source (comediator-fiber ?j)) (cocone-fiber ?j))
```

```
(= (SET.FTN$target (comediator-fiber ?j)) set.mor$morphism)
(= (SET.FTN$composition [(comediator-fiber ?j) set.mor$source])
   (SET.FTN$composition [(cocone-tuple-fiber ?j) (colimit-fiber ?j)]))
(= (SET.FTN$composition [(comediator-fiber ?j) set.mor$target])
   (opvertex-fiber ?j))
(KIF$restriction (comediator-fiber ?j) comediator)))
```

Coproducts of Tuple Morphisms

`set.col.coprd.mor`

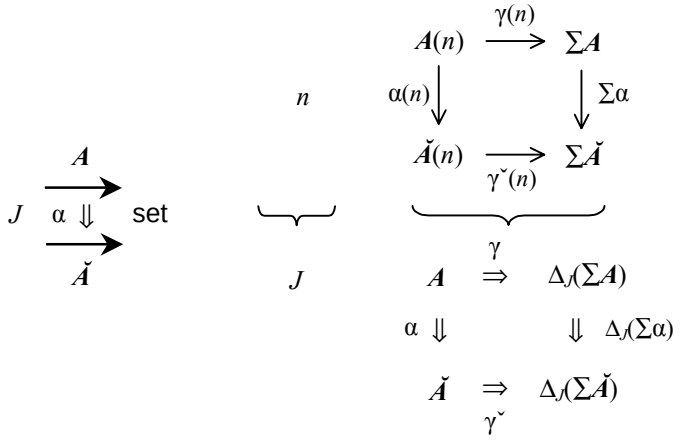


Diagram 28: Tuple Morphisms and Coproducts

Coproducts are defined on tuple morphisms. Tuple morphisms offer a change of perspective for coproducts. The coproduct function maps a set tuple morphism $\boldsymbol{\alpha} : \boldsymbol{A} \Rightarrow \check{\boldsymbol{A}} : J \to \mathsf{set}$ (Diagram 28) to a coproduct set function $\boldsymbol{\Sigma}\alpha : \boldsymbol{\Sigma A} \to \boldsymbol{\Sigma}\check{\boldsymbol{A}}$. As such, it is the morphism function of a coproduct (quasi)functor $\boldsymbol{\Sigma} : \mathsf{Set}^J \to \mathsf{Set}$ from the functor (quasi)category Set^J to Set, where the object function is given by the coproduct function on set tuples. The coproduct of a tuple morphism is defined as the comediator of a morphism cocone. The coproduct operator preserves composition and identities.

```
(1) (SET.FTN$function morphism-cocone)
    (= (SET.FTN$source morphism-cocone) set.tpl.mor$morphism)
    (= (SET.FTN$target morphism-cocone) set.col.coprd$cocone)
    (= (SET.FTN$composition [morphism-cocone set.col.coprd$cocone-tuple])
        set.tpl.mor$source)
    (= (SET.FTN$composition [morphism-cocone set.col.coprd$opvertex])
       (SET.FTN$composition [set.dgm.mor$target set.col.coprd$coproduct]))
    (forall (?a (set.dgm.tpl.mor$morphism ?a)
                ?n ((set.dgm.tpl.mor$index ?a) ?n))
        (= ((set.col.coprd$component (morphism-cocone ?a)) ?n)
           (set.ftn$composition
               [((set.dgm.tpl.mor$component ?a) ?n)
                ((set.col.coprd$injection (set.dgm.tpl.mor$target ?a)) ?n)])))

(2) (KIF$function coproduct)
    (= (KIF$source coproduct) set.dgm.mor$morphism)
    (= (KIF$target coproduct) set.col$cocone)
    (forall (?a (set.dgm.mor$morphism ?a))
        (and (= (set.mor$source (coproduct ?a))
                (set.col$colimit (set.dgm.tpl.mor$source ?a)))
             (= (set.mor$target (coproduct ?a))
                (set.col$colimit (set.dgm.tpl.mor$target ?a)))
             (= (coproduct ?a) (set.col.coprd$comediator (morphism-cocone ?a)))))

(3) (forall (?a (set.dgm.tpl.mor$morphism ?a)
                ?b (set.dgm.tpl.mor$morphism ?b)
               (set.dgm.tpl.mor$composable ?a ?b))
         (= (coproduct (set.dgm.tpl.mor$composition [?a ?b]))
            (set.ftn$composition [(coproduct ?a) (coproduct ?b)])))

(4) (forall (?a (set.dgm.tpl$tuple ?a))
        (= (coproduct (set.dgm.tpl.mor$identity ?a))
           (set.mor$identity (set.col.coprd$coproduct ?a))))
```

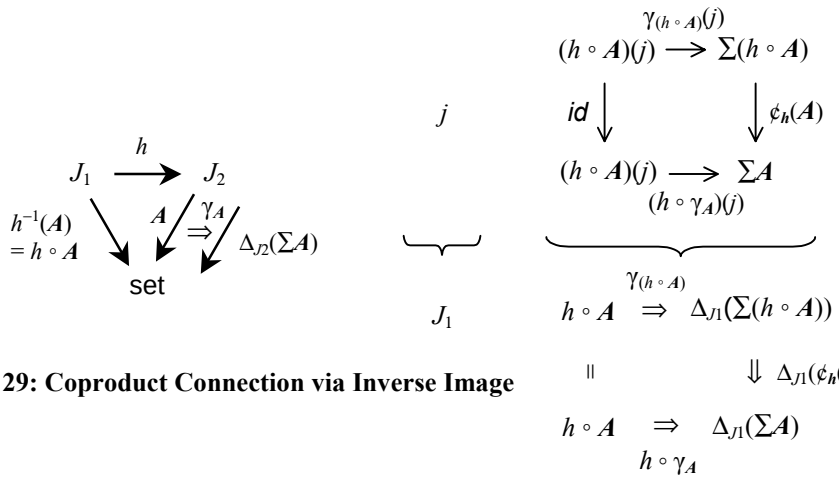


Diagram 29: Coproduct Connection via Inverse Image

The following operator is needed in order to define coproducts on colax tuple morphisms (Diagram 29). The coproduct of a tuple is connected to the coproduct of its inverse image along a set function. For any set function $h : J_1 \to J_2$, the *connection* function maps a tuple $\boldsymbol{A} : J_2 \to \mathsf{set}$ in the target fiber to a connection set function $\phi_h(\boldsymbol{A}) : \Sigma h^{-1}(\boldsymbol{A}) = \Sigma(h \circ \boldsymbol{A}) \to \Sigma\boldsymbol{A}$ from the coproduct of the inverse image tuple $h^{-1}(\boldsymbol{A}) = (h \circ \boldsymbol{A}) : J_1 \to \mathsf{set}$ to the coproduct of $\boldsymbol{A}$. This function is defined as the comediator of a connection cocone.

```
(5) (KIF$function connection-cocone)
    (= (KIF$source connection-cocone) set.mor$morphism)
    (= (KIF$target connection-cocone) SET.FTN$function)
    (forall (?h (set.mor$morphism ?h))
        (and (= (SET.FTN$source (connection-cocone ?h))
                (set.dgm.tpl$tuple-fiber (set.mor$target ?h)))
             (= (SET.FTN$target (connection-cocone ?h))
                (set.col.coprd$cocone-fiber (set.mor$source ?h)))
             (= (SET.FTN$composition
                    [(connection-cocone ?h)
                     (set.col.coprd$cocone-tuple-fiber (set.mor$source ?h))])
                (inverse-image ?h))
             (= (SET.FTN$composition
                    [(connection-cocone ?h)
                     (set.col.coprd$opvertex-fiber (set.mor$source ?h))])
                (set.col.coprd$colimit-fiber (set.mor$target ?h)))
             (forall (?a ((tuple-fiber (set.mor$target ?h)) ?a))
                 (= ((set.col.coprd$component-fiber (set.mor$target ?h))
                        ((connection-cocone ?h) ?a))
                    (SET.FTN$composition
                        [?h ((set.col.coprd$injection-fiber (set.mor$target ?h)) ?a)]))))))

(6) (KIF$function connection)
    (= (KIF$source connection) set.mor$morphism)
    (= (KIF$target connection) SET.FTN$function)
    (forall (?h (set.mor$morphism ?h))
        (and (= (SET.FTN$source (connection ?h))
                (set.dgm.coprd$tuple-fiber (set.mor$target ?h)))
             (= (SET.FTN$target (connection ?h)) set.mor$morphism)
             (= (SET.FTN$composition [(connection ?h) set.mor$source])
                (SET.FTN$composition
                    [(inverse-image ?h) (set.col.coprd$colimit-fiber (set.mor$source ?h))]))
             (= (SET.FTN$composition [(connection ?h) set.mor$target])
                (set.col.coprd$colimit-fiber (set.mor$target ?h)))
             (= (connection ?h)
                (SET.FTN$composition
                    [((connection-cocone ?h)
                     (set.col.coprd$comediator-fiber (set.mor$source ?h))])))))
```

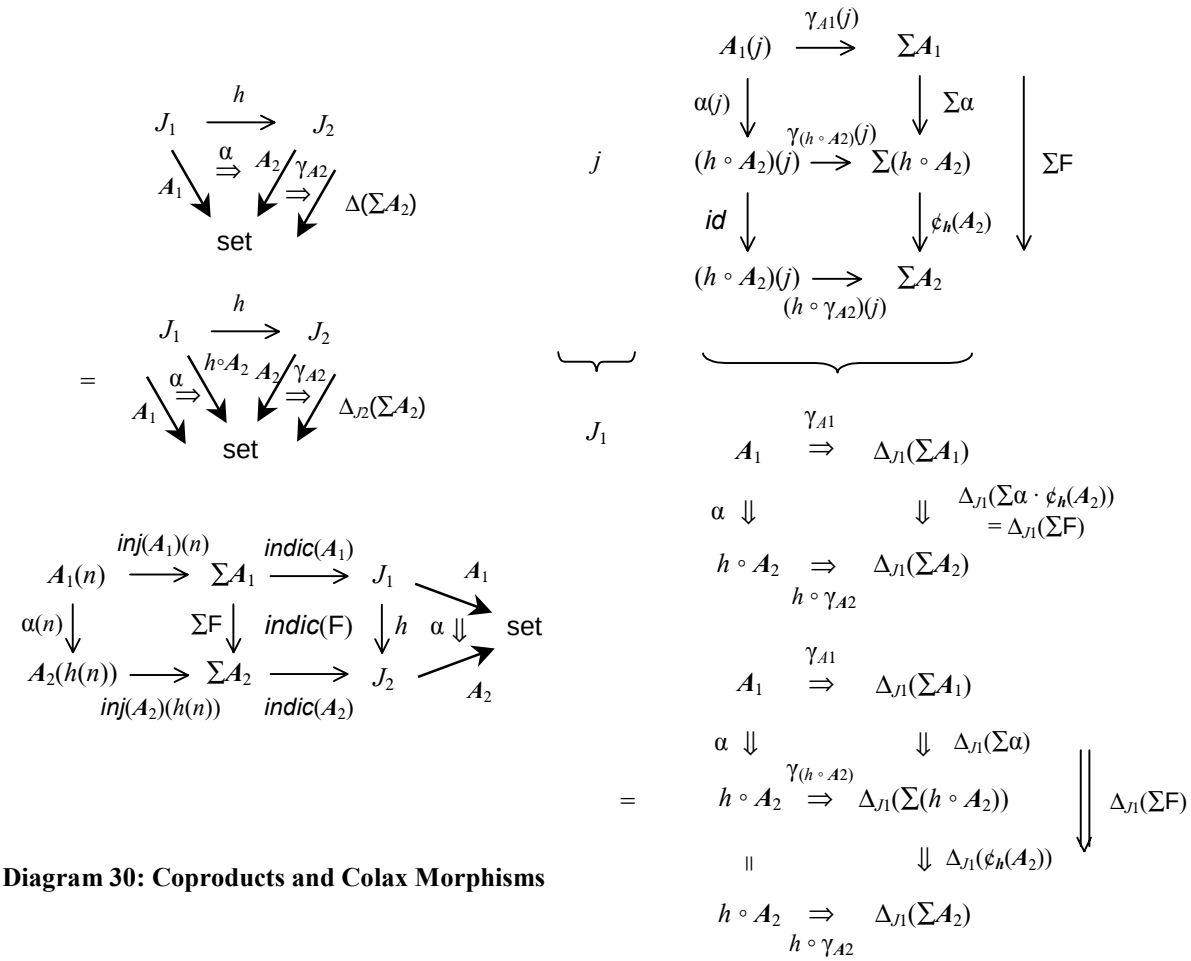


Diagram 30: Coproducts and Colax Morphisms

We put the previous two ideas together here (Diagram 30). Coproducts are defined on colax tuple morphisms. Colax tuple morphisms offer a change of perspective for coproducts. The *colax-coproduct* function maps a set tuple colax morphism $\mathsf{F} = (h, \alpha) : (J_1, \boldsymbol{A}_1) \Rightarrow (J_2, \boldsymbol{A}_2)$ to a colax-coproduct set function $\Sigma\mathsf{F} : \Sigma\boldsymbol{A}_1 \to \Sigma\boldsymbol{A}_2$. As such, it is the morphism function of a colimit (quasi)functor $\Sigma : \mathsf{Cat} \Uparrow \mathsf{Set} \to \mathsf{Set}$ from the "super-comma" (quasi)category with objects being set tuples and morphisms being colax tuple morphisms. Here, the object function is given by the colimit function on set tuples. The coproduct of a colax tuple morphism is the composition of the coproduct of the tuple morphism $\boldsymbol{\alpha} : \boldsymbol{A}_1 \Rightarrow h \circ \boldsymbol{A}_2 : J \to \mathsf{set}$ followed by the connection of the target tuple $\boldsymbol{A}_2 : J_2 \to \mathsf{set}$. The colax coproduct operator preserves composition and identities.

```
(7) (KIF$function colax-colimit)
    (= (KIF$source colax-colimit) set.dgm.tpl.clmor$colax-morphism)
    (= (KIF$target colax-colimit) set.mor$morphism)
    (forall (?f (set.dgm.tpl.clmor$colax-morphism ?f))
        (and (= (set.mor$source (colax-colimit ?f))
                (set.col.coprd$colimit (set.dgm.tpl.clmor$source ?f)))
             (= (set.mor$target (colax-colimit ?f))
                (set.col.coprd$colimit (set.dgm.tpl.clmor$target ?f)))
             (= (colax-colimit ?f)
                (set.mor$composition
                    [(colimit ?a)
                     ((connection (set.dgm.tpl.clmor$set-function ?f))
                         (set.dgm.tpl.clmor$target ?f))])))))

(8) (forall (?f (set.dgm.tpl.clmor$morphism ?f)
             ?g (set.dgm.tpl.clmor$morphism ?g)
            (set.dgm.tpl.clmor$composable ?f ?g))
```

```
        (= (colimit (set.dgm.tpl.clmor$composition [?f ?g]))
           (set.ftn$composition [(colimit ?f) (colimit ?g)])))

(9) (forall (?a (set.dgm.tpl$tuple ?a))
       (= (colimit (set.dgm.tpl.clmor$identity ?a))
          (set.mor$identity (set.col.coprd$colimit ?a))))
```

When the coproducts are standard, the *standard* pointwise definition of the *colax coproduct* is:

$$\sum \mathsf{F}((j, x)) = (h(j), \alpha(j)(x))$$

for any index $j \in J_1$ and element $x \in \boldsymbol{A}(j)$. This is well defined by definition of the morphism α. The colax coproduct function is the cotupling of the injection of components:

$$\{\alpha(j) \cdot \mathit{inj}(\boldsymbol{B})(h(j)) \mid j \in \mathit{ind}(\boldsymbol{A})\}.$$

For any colax morphism of tuple sets $\mathsf{F} = (h, \alpha) = \langle \mathit{ind}(\mathsf{F}), \mathit{mor}(\mathsf{F}) \rangle : (J_1, \boldsymbol{A}_1) \Rightarrow (J_2, \boldsymbol{A}_2)$ the coproduct function is the vertical source for an *indication* quartet $\mathit{indic}(\boldsymbol{h})$. The commutativity (preservation of indication) $\sum \mathsf{F} \cdot \mathit{indic}(\boldsymbol{B}) = \mathit{indic}(\boldsymbol{A}) \cdot h$, is a property of the coproduct of tuple sets. This is obvious from the pointwise definition of the coproduct function.

```
(10) (KIF$function indication)
     (= (KIF$source indication) set.dgm.tpl.clmor$morphism)
     (= (KIF$target indication) set.qtt$quartet)
     (forall (?f (set.dgm.tpl.clmor$morphism ?f))
         (and (= (set.qtt$horizontal-source (indication ?f))
                 (set.col.coprd$indication (source ?f)))
              (= (set.qtt$horizontal-target (indication ?f))
                 (set.col.coprd$index (target ?f)))
              (= (set.qtt$vertical-source (indication ?f)) (colax-colimit ?f))
              (= (set.qtt$vertical-target (indication ?f)) (index ?f))))
```

Coproducts of Arities

`set.col.art`

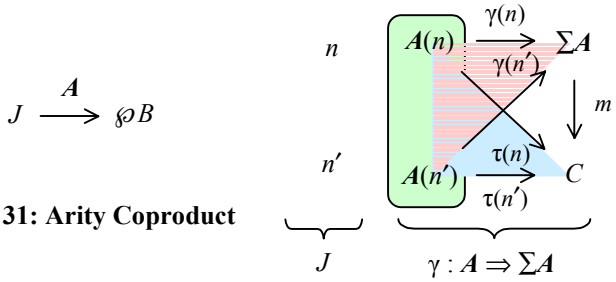


Diagram 31: Arity Coproduct

A *cocone* over arity $\boldsymbol{A} : J \to \wp B$ (Diagram 31), consists of an *opvertex* C, and a collection τ of *component* functions indexed by the nodes in the index set of the arity. The cocone is situated over the arity.

```
(1) (SET$class cocone)

(2) (SET.FTN$function cocone-arity)
    (= (SET.FTN$source cocone-arity) cocone)
    (= (SET.FTN$target cocone-arity) set.dgm.art$arity)

(3) (SET.FTN$function opvertex)
    (= (SET.FTN$source opvertex) cocone)
    (= (SET.FTN$target opvertex) set.obj$object)

(4) (KIF$function component)
    (= (KIF$source component) cocone)
    (= (KIF$target component) SET.FTN$function)
    (forall (?s (cocone ?s))
        (and (= (SET.FTN$source (component ?s)) (set.dgm.art$index (cocone-arity ?s)))
             (= (SET.FTN$target (component ?s)) set.mor$morphism)
             (= (SET.FTN$composition [(component ?s) set.mor$source])
                (set.dgm.art$set (cocone-arity ?s)))
             (= (SET.FTN$composition [(component ?s) set.mor$target])
                ((set.ftn$constant [(set.dgm.art$index (cocone-arity ?s)) set.obj$object])
                    (opvertex ?s)))))
```

The *colimiting cocone* function maps an arity $\boldsymbol{A} : J \to \wp B$ to its colimiting coproduct cocone (Diagram 32)

$$\mathit{coprd}(\boldsymbol{A}) = \textstyle\sum_{n \in \mathit{ind}(\boldsymbol{A})} \boldsymbol{A}(n).$$

A colimiting coproduct cocone is the special case of a general colimiting cocone over a coproduct diagram (discrete diagram or tuple).

For any arity $\boldsymbol{A}$ and any index $n \in J = \mathit{ind}(\boldsymbol{A})$ there is an *injection* function

$$\mathit{inj}(\boldsymbol{A})(n) : \boldsymbol{A}(n) \to \mathit{coprd}(\boldsymbol{A}).$$

Injections are injective. They commute with projection and inclusion.

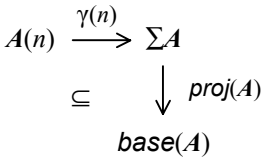


Diagram 32: Coproduct

```
(5) (SET.FTN$function colimiting-cocone)
    (= (SET.FTN$source colimiting-cocone) set.dgm.art$arity)
    (= (SET.FTN$target colimiting-cocone) cocone)
    (= (SET.FTN$composition [colimiting-cocone cocone-arity])
       (SET.FTN$identity set.dgm.art$arity))

(6) (SET.FTN$function colimit)
    (SET.FTN$function coproduct)
    (= coproduct colimit)
    (= (SET.FTN$source colimit) set.dgm.art$arity)
    (= (SET.FTN$target colimit) set.obj$object)
    (= colimit (SET.FTN$composition [colimiting-cocone opvertex]))

(7) (KIF$function injection)
    (= (KIF$source injection) set.dgm.art$arity)
    (= (KIF$target injection) SET.FTN$function)
    (forall (?a (set.dgm.art$arity ?a)
```

```
(= (injection ?a) (component (colimiting-cocone ?a))))
```

For the convenience of the implementer, we also provide a standard coproduct and a standard collection of injections functions that she may choose. The *standard coproduct* or *disjoint union* of an arity

$$\mathit{std\text{-}coprd}(\boldsymbol{A}) = \textstyle\sum_{n \in J} \boldsymbol{A}(n)$$
$$= \{(n, x) \mid n \in J, x \in \boldsymbol{A}(n)\}$$

provides a concrete coproduct for that arity. For any arity $\boldsymbol{A} = \{\boldsymbol{A}(n) \mid n \in J, \boldsymbol{A}(n) \subseteq B\}$ and any index $n \in J = \mathit{ind}(\boldsymbol{A})$ there is a *standard injection* function

$$\mathit{std\text{-}inj}(\boldsymbol{A})(n) : \boldsymbol{A}(n) \to \mathit{std\text{-}coprd}(\boldsymbol{A})$$

defined by $\mathit{std\text{-}inj}(\boldsymbol{A})(n)(x) = (n, x)$ for all indices $n \in J = \mathit{ind}(\boldsymbol{A})$ and all elements $x \in \boldsymbol{A}(n)$. Obviously, the standard injections are injective. The *standard colimiting cocone* is then constructed out of the standard coproduct and the standard injection functions. Note how this proceeds bottom-up and in reverse order to the specific components.

```
 (8) (SET.FTN$function standard-colimit)
     (SET.FTN$function standard-coproduct)
     (SET.FTN$function disjoint-union)
     (= standard-coproduct standard-colimit)
     (= disjoint-union standard-coproduct)
     (= (SET.FTN$source standard-colimit) set.dgm.art$arity)
     (= (SET.FTN$target standard-colimit) set.obj$object)
     (forall (?a (set.dgm.art$arity ?a))
         (<=> ((standard-colimit ?a) ?x)
              (exists (?n ((set.dgm.art$index ?a) ?n)
                       ?xn (((set.dgm.art$set ?a) ?n) ?xn))
                  (= ?x [?n ?xn])))))

 (9) (SET.FTN$function standard-injection)
     (= (SET.FTN$source standard-injection) set.dgm.art$arity)
     (= (SET.FTN$target standard-injection) SET.FTN$function)
     (forall (?a (set.dgm.art$arity ?a)
              ?n ((set.dgm.art$index ?a) ?n)
              ?xn (((set.dgm.art$set ?a) ?n) ?xn))
         (= (((standard-injection ?a) ?n) ?xn) [?n ?xn]))

(10) (SET.FTN$function standard-colimiting-cocone)
     (= (SET.FTN$source standard-colimiting-cocone) set.dgm.art$arity)
     (= (SET.FTN$target standard-colimiting-cocone) cocone)
     (= (SET.FTN$composition [standard-colimiting-cocone cocone-arity])
        (SET.FTN$identity set.dgm.art$arity))
     (= (SET.FTN$composition [standard-colimiting-cocone opvertex]) standard-coproduct)
     (= (SET.FTN$composition [standard-colimiting-cocone component]) standard-injection)
```

There is a *comediator* function (Diagram 32) from the coproduct of the arity of a cocone to the opvertex of the cocone. This is the unique function that commutes with the component functions of the cocone. We have also introduced a "convenience term" cotupling. With an arity parameter, this maps a tuple of set functions, which form a coproduct cocone with the arity, to their comediator (or *cotupling*) function.

```
(11) (SET.FTN$function comediator)
     (= (SET.FTN$source comediator) cocone)
     (= (SET.FTN$target comediator) set.mor$morphism)
     (forall (?s (cocone ?s))
        (= (comediator ?s)
           (the (?m (set.mor$morphism ?m))
                (and (= (set.mor$source ?m) (colimit (cocone-arity ?s)))
                     (= (set.mor$target ?m) (opvertex ?s))
                     (forall (?n ((index (cocone-arity ?s)) ?n))
                         (= (set.mor$composition [((injection (cocone-arity ?s)) ?n) ?m])
                            ((component ?s) ?n)))))))))

(12) (KIF$function cotupling-cocone)
     (KIF$source cotupling-cocone) set.dgm.art$arity)
     (KIF$target cotupling-cocone) SET.FTN$partial-function)
     (forall (?a (set.dgm.art$arity ?a))
         (and (= (SET.FTN$source (cotupling-cocone ?a))
                 (SET.FTN$power [(set.dgm.art$index ?a) set.mor$morphism]))
              (= (SET.FTN$target (cotupling-cocone ?a)) cocone)
```

```
            (forall (?f ((SET.FTN$power [(set.dgm.art$index ?a) set.mor$morphism]) ?f))
                (<=> ((SET.FTN$domain (cotupling-cocone ?a)) ?f)
                     (and (forall (?n ((set.dgm.art$index ?a) ?n))
                              (= (set.mor$source (?f ?n)) ((set.dgm.art$set ?a) ?n)))
                          (exists (?c (set.obj$object ?c))
                              (forall (?n ((set.dgm.art$index ?a) ?n))
                                  (= (set.mor$target (?f ?n)) ?c)))))))

(13) (KIF$function cotupling)
     (= (KIF$source cotupling) set.dgm.art$arity)
     (= (KIF$target cotupling) SET.FTN$partial-function)
     (forall (?a (set.dgm.art$arity ?a))
         (and (= (SET.FTN$source (cotupling ?a))
                 (SET.FTN$power [(set.dgm.art$index ?a) set.mor$morphism]))
              (= (SET.FTN$target (cotupling ?a)) set.mor$morphism)
              (= (SET.FTN$domain (cotupling ?a))
                 (SET.FTN$domain (cotupling-cocone ?a)))
              (forall ?f ((SET.FTN$domain (cotupling ?a)) ?f))
                  (= ((cotupling ?a) ?f)
                     (comediator ((cotupling-cocone ?a) ?f))))))
```

There is an *indication* function (special for coproducts over tuples and arities) (Diagram 33)

$indic(\boldsymbol{A}) : \sum \boldsymbol{A} \to J = ind(\boldsymbol{A})$

from the coproduct to the arity index. The indication function is the cotupling of the collection of constant index functions

$\{\Delta_{\boldsymbol{A}}(n) : \boldsymbol{A}(n) \to ind(\boldsymbol{A}) \mid n \in ind(\boldsymbol{A})\}$.

There is a *projection* function (special for coproducts over arities) (Diagram 33)

$proj(\boldsymbol{A}) : \sum \boldsymbol{A} \to base(\boldsymbol{A})$

from the coproduct to the arity base. The projection function is the cotupling of the collection of inclusion functions

$\{incl_{\boldsymbol{A}}(n) : \boldsymbol{A}(n) \to base(\boldsymbol{A}) \mid n \in ind(\boldsymbol{A})\}$.

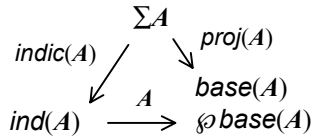


Diagram 33: Colimit, and Indication and Projection Functions of an Arity

```
(14) (KIF$function constant-index)
     (= (KIF$source constant-index) set.dgm.art$arity)
     (= (KIF$target constant-index) SET.FTN$function)
     (forall (?a (set.dgm.art$arity ?a))
         (and (= (SET.FTN$source (constant-index ?a)) (set.dgm.art$index ?a))
              (= (SET.FTN$target (constant-index ?a)) set.mor$morphism)
              (= (SET.FTN$composition [(constant-index ?a) set.mor$source]) (set.dgm.art$set ?a))
              (= (SET.FTN$composition [(constant-index ?a) set.mor$target])
                 ((SET.FTN$constant [(set.dgm.art$index ?a) set.obj$object]) (set.dgm.art$index ?a)))
              (forall (?n ((set.dgm.art$index ?a) ?n))
                  (= ((constant-index ?a) ?n)
                     ((set.mor$constant [((set.dgm.art$set ?a) ?n) (set.dgm.art$index ?a)]) ?n))))))

(15) (SET.FTN$function indication)
     (= (SET.FTN$source indication) set.dgm.art$arity)
     (= (SET.FTN$target indication) set.mor$morphism)
     (= (SET.FTN$composition [indication set.mor$source]) coproduct)
     (= (SET.FTN$composition [indication set.mor$target]) set.dgm.art$index)
     (forall (?a (set.dgm.art$arity ?a))
         (= (indication ?a) ((cotupling ?a) (constant-index ?a)))))

(16) (KIF$function inclusion)
     (= (KIF$source inclusion) set.dgm.art$arity)
     (= (KIF$target inclusion) SET.FTN$function)
     (forall (?a (set.dgm.art$arity ?a))
         (and (= (SET.FTN$source (inclusion ?a)) (set.dgm.art$index ?a))
              (= (SET.FTN$target (inclusion ?a)) set.mor$morphism)
              (= (SET.FTN$composition [(inclusion ?a) set.mor$source]) (set.dgm.art$set ?a))
              (= (SET.FTN$composition [(inclusion ?a) set.mor$target])
                 ((SET.FTN$constant [(set.dgm.art$index ?a) set.obj$object]) (set.dgm.art$base ?a)))
              (forall (?n ((set.dgm.art$index ?a) ?n))
                  (= ((inclusion ?a) ?n)
                     (set.mor$inclusion [((set.dgm.art$set ?a) ?n) (set.dgm.art$base ?a)])))))
```

```
(17) (SET.FTN$function projection)
     (= (SET.FTN$source projection) set.dgm.art$arity)
     (= (SET.FTN$target projection) set.mor$morphism)
     (= (SET.FTN$composition [projection set.mor$source]) coproduct)
     (= (SET.FTN$composition [projection set.mor$target]) set.dgm.art$base)
     (forall (?a (set.dgm.art$arity ?a))
         (= (projection ?a) ((cotupling ?a) (inclusion ?a))))
```

For the convenience of the implementer, we also provide standard indication and projection functions that she may choose. The *standard indication* function

$\mathit{std\text{-}indic}(\boldsymbol{A}) : \mathit{std\text{-}coprd}(\boldsymbol{A}) \to J = \mathit{ind}(\boldsymbol{A})$

from the standard coproduct to the arity index is defined by $\mathit{std\text{-}indic}(\boldsymbol{A})(n, x) = n$.

The *standard projection* function

$\mathit{std\text{-}proj}(\boldsymbol{A}) : \mathit{std\text{-}coprd}(\boldsymbol{A}) \to \mathit{base}(\boldsymbol{A})$

from the standard coproduct to the arity base is defined by $\mathit{std\text{-}proj}(\boldsymbol{A})(n, x) = x$.

```
(18) (SET.FTN$function standard-indication)
     (= (SET.FTN$source standard-indication) set.dgm.art$arity)
     (= (SET.FTN$target standard-indication) set.mor$morphism)
     (= (SET.FTN$composition [standard-indication set.mor$source]) standard-coproduct)
     (= (SET.FTN$composition [standard-indication set.mor$target]) set.dgm.art$index)
     (forall (?a (set.dgm.art$arity ?a)
              ?n ((set.dgm.art$index ?a) ?n)
              ?xn (((set.dgm.art$set ?a) ?n) ?xn))
         (= ((standard-indication ?a) [?n ?xn]) ?n))

(19) (SET.FTN$function standard-projection)
     (= (SET.FTN$source standard-projection) set.dgm.art$arity)
     (= (SET.FTN$target standard-projection) set.mor$morphism)
     (= (SET.FTN$composition [standard-projection set.mor$source]) standard-coproduct)
     (= (SET.FTN$composition [standard-projection set.mor$target]) set.dgm.art$base)
     (forall (?a (set.dgm.art$arity ?a)
              ?n ((set.dgm.art$index ?a) ?n)
              ?xn (((set.dgm.art$set ?a) ?n) ?xn))
         (= ((standard-projection ?a) [?n ?xn]) ?xn))
```

Coproducts of Arity Morphisms

`set.col.art.mor`

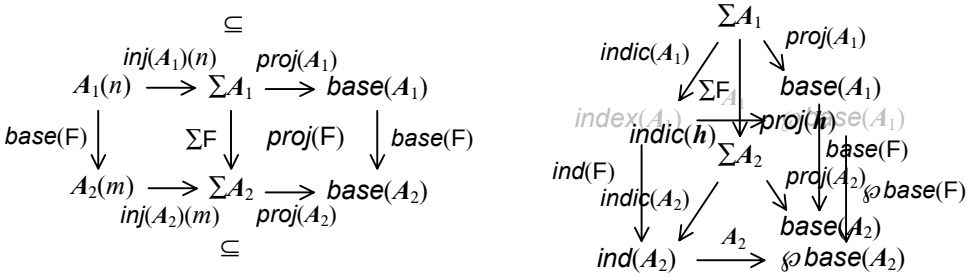


Diagram 34: Arity Morphism Coproduct

Any morphism of arities $\mathsf{F} = \langle \mathit{ind}(\mathsf{F}), \mathit{base}(\mathsf{F}) \rangle : \boldsymbol{A}_1 \to \boldsymbol{A}_2$ defines a *coproduct* function (Diagram 34)

$\sum \mathsf{F} : \sum \boldsymbol{A}_1 \to \sum \boldsymbol{A}_2$.

When the coproducts are standard, the pointwise definition is:

$\sum \mathsf{F}((n, x)) = (\mathit{ind}(\mathsf{F})(n), \mathit{base}(\mathsf{F})(x))$

for any index $n \in \mathit{ind}(\boldsymbol{A}_1)$ and element $x \in \boldsymbol{A}_1(n)$. This is well defined since F preserves index arity. The coproduct function is the cotupling of the injection of *base restrictions*:

$\{\mathit{base}(\boldsymbol{h}) \cdot \mathit{inj}(\boldsymbol{A}_2)(\mathit{ind}(\mathsf{F})(n)) \mid n \in \mathit{ind}(\boldsymbol{A}_1)\}$.

```
(1) (KIF$function base-restriction)
    (= (KIF$source base-restriction) set.dgm.art.mor$morphism)
    (= (KIF$target base-restriction) SET.FTN$function)
    (forall (?f (set.dgm.art.mor$morphism ?f))
        (and (= (SET.FTN$source (base-restriction ?f)) (set.dgm.art$index (source ?f)))
             (= (SET.FTN$target (base-restriction ?f)) set.mor$morphism)
             (= (SET.FTN$composition [(base-restriction ?f) set.mor$source])
                (set.dgm.art$set (source ?f)))
             (= (SET.FTN$composition [(base-restriction ?f) set.mor$target])
                (set.dgm.art$set (target ?f)))
             (forall (?n ((set.dgm.art$index (source ?f)) ?n))
                 (set.mor$restriction
                     ((base-restriction ?f) ?n)
                     (set.dgm.art.mor$base ?f)))))

(2) (KIF$function coproduct-tuple)
    (= (KIF$source coproduct-tuple) set.dgm.art.mor$morphism)
    (= (KIF$target coproduct-tuple) SET.FTN$function)
    (forall (?f (set.dgm.art.mor$morphism ?f))
        (and (= (SET.FTN$source (coproduct-tuple ?f)) (set.dgm.art$index (source ?f)))
             (= (SET.FTN$target (coproduct-tuple ?f)) set.mor$morphism)
             (= (SET.FTN$composition [(coproduct-tuple ?f) set.mor$source])
                (set.dgm.art$set (source ?f)))
             (= (SET.FTN$composition [(coproduct-tuple ?f) set.mor$target])
                (set.col.art$coproduct (target ?f)))
             (forall (?n ((set.dgm.art$index (source ?f)) ?n))
                 (= ((coproduct-tuple ?f) ?n)
                    (set.mor$composition
                        [(base-restriction ?f) ?n)
                         ((set.col.art$injection (target ?f)) ((index ?f) ?n))])))))

(3) (SET.FTN$function coproduct)
    (= (SET.FTN$source coproduct) set.dgm.art.mor$morphism)
    (= (SET.FTN$target coproduct) set.mor$morphism)
    (= (SET.FTN$composition [coproduct set.mor$source])
       (SET.FTN$composition [source set.col.art$coproduct]))
    (= (SET.FTN$composition [coproduct set.mor$target])
       (SET.FTN$composition [target set.col.art$coproduct]))
```

```
(forall (?f (set.dgm.art.mor$morphism ?f))
    (= (coproduct ?f)
       ((cotupling (source ?f)) (coproduct-tuple ?f))))
```

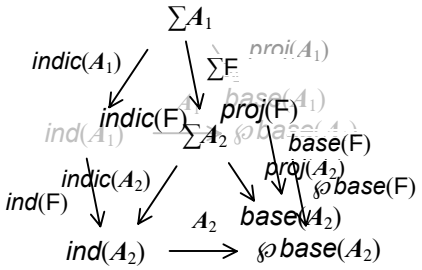


Figure 34: Coproduct of the Diagram of an Arity Morphism

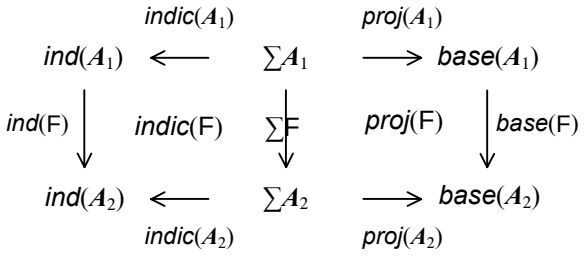


Figure 35: Spangraph Morphism of a Arity Morphism

For any morphism of arities $\mathsf{F} = \langle \mathit{ind}(\mathsf{F}), \mathit{base}(\mathsf{F}) \rangle : \boldsymbol{A}_1 \to \boldsymbol{A}_2$ the coproduct function (Diagrams 34, 35) is the vertical source for two quartets: an *indication* quartet $\mathit{indic}(\boldsymbol{h})$ and a *projection* quartet $\mathit{proj}(\boldsymbol{h})$.

– The commutativity (preservation of indication)

$$\textstyle\sum \mathsf{F} \cdot \mathit{indic}(\boldsymbol{A}_2) = \mathit{indic}(\boldsymbol{A}_1) \cdot \mathit{ind}(\mathsf{F}),$$

is a property of the coproduct of arities. This is obvious from the pointwise definition of the coproduct function. This is provable using coproduct cotupling.

– The commutativity (preservation of projection)

$$\textstyle\sum \mathsf{F} \cdot \mathit{proj}(\boldsymbol{A}_2) = \mathit{proj}(\boldsymbol{A}_1) \cdot \mathit{base}(\mathsf{F}),$$

is also a property of the coproduct of arities. This is obvious from the pointwise definition of the coproduct function. This is also provable using coproduct cotupling.

By the preservation of index arity, $\wp\mathit{base}(\mathsf{F})(\boldsymbol{A}_1(n)) = \boldsymbol{A}_2(\mathit{ind}(\mathsf{F})(n))$ for every source index $n \in \mathit{ind}(\boldsymbol{A}_1)$, the index quartet is a fibration: for any index $n \in \mathit{ind}(\boldsymbol{A}_1)$ and any element $y \in \boldsymbol{A}_2(\mathit{ind}(\mathsf{F})(n))$ there is an element $x \in \boldsymbol{A}_1(n)$ such that $\mathit{base}(\mathsf{F})(x) = y$.

```
(4) (SET.FTN$function indication)
    (= (SET.FTN$source indication) set.dgm.art.mor$morphism)
    (= (SET.FTN$target indication) set.qtt$fibration)
    (= (SET.FTN$composition [indication set.qtt$horizontal-source])
       (SET.FTN$composition [source set.col.art$indication]))
    (= (SET.FTN$composition [indication set.qtt$horizontal-target])
       (SET.FTN$composition [target set.col.art$indication]))
    (= (SET.FTN$composition [indication set.qtt$vertical-source]) coproduct)
    (= (SET.FTN$composition [indication set.qtt$vertical-target]) index)

(5) (SET.FTN$function projection)
    (= (SET.FTN$source projection) set.dgm.art.mor$morphism)
    (= (SET.FTN$target projection) set.qtt$quartet)
    (= (SET.FTN$composition [projection set.qtt$horizontal-source])
       (SET.FTN$composition [source set.col.art$projection]))
    (= (SET.FTN$composition [projection set.qtt$horizontal-target])
       (SET.FTN$composition [target set.col.art$projection]))
    (= (SET.FTN$composition [projection set.qtt$vertical-source]) coproduct)
    (= (SET.FTN$composition [projection set.qtt$vertical-target]) base)
```

Binary Coproducts

set.col.coprd2

This section is crucial.

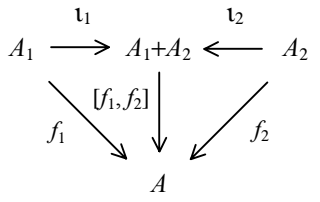


Figure 36: Binary Coproduct Set, Cocone and Comediator

A *binary coproduct* is a finite colimit in the category Set for a diagram of shape *two* = • •. Such a diagram (of sets and set functions) is called a *pair* of sets. Given a pair of set A_1 and A_2, a particular *binary coproduct* or *sum* $A_1 + A_2$ (Figure 36) is the disjoint union $A_1 \oplus A_2 = \{(1, a_1) \mid a_1 \in A_1\} \cup \{(2, a_2) \mid a_2 \in A_2\}$. That is, elements of the disjoint union are either tagged elements $(1, a_1)$ where $a_1 \in A_1$ or tagged elements $(2, a_2)$ where $a_2 \in A_2$. The *sum injection* set function $\iota_1 = in(A_1) : A_1 \to A_1 \oplus A_2$ from the first component set A_1 to the disjoint union set $A_1 \oplus A_2$ is maps an element $a_1 \in A_1$ to the tagged element $(1, a_1) \in A_1 + A_2$. To show universality, let $f_1 : A_1 \to A$ and $f_2 : A_2 \to A$ be any functions from the components to a common target set A. We need to show that there is a unique set function $[f_1, f_2] : A_1 \to A$ called the comediator such that $\iota_1 \cdot [f_1, f_2] = f_1$ and $\iota_2 \cdot [f_1, f_2] = f_2$. By necessity, $[f_1, f_2](1, a_1) = [f_1, f_2](\iota_1(a_1)) = f_1(a_1)$ for a tagged element $(1, a_1) \in A_1 + A_2$ and $[f_1, f_2](2, a_2) = [f_1, f_2](\iota_2(a_2)) = f_2(a_2)$ for a tagged element $(1, a_2) \in A_1 \oplus A_2$. Let this be the definition of the comediator $[f_1, f_2]$. The disjoint union $A_1 \oplus A_2$ is a binary coproduct in the category Set. As discussed for general colimits above, we do not require that the disjoint union be used for the binary coproduct, but only use a "moderate expression" consisting of the description of a map from any pair of sets to a colimiting cocone, with axiomatized assertions that this is a binary coproduct for the pair. Since the disjoint union is a binary coproduct, and all binary coproducts are isomorphic, the chosen binary coproduct is guaranteed to be isomorphic to the disjoint union. However, it is not required to be equal to it.

A *binary coproduct cocone* is the appropriate cocone for a binary coproduct. A binary coproduct cocone (see Figure 36, where arrows denote functions) consists of a pair of set functions called *opfirst* and *opsecond*. These are required to have a common target set called the *opvertex* of the cocone. Each binary coproduct cocone is over a pair of sets. A binary coproduct cocone is the very special case of a cocone over a pair of sets. We connect this specific terminology to the generic fiber terminology for set colimits.

```
(1) (SET$class cocone)
    (= cocone (set.col$cocone-fiber gph$two)

(2) (SET.FTN$function cocone-diagram)
    (= (SET.FTN$source cocone-diagram) cocone)
    (= (SET.FTN$target cocone-diagram) set.dgm.pr$diagram)
    (= cocone-diagram (set.col$cocone-tuple-fiber gph$two))

(3) (SET.FTN$function opvertex)
    (= (SET.FTN$source opvertex) cocone)
    (= (SET.FTN$target opvertex) set.obj$object)
    (= opvertex (set.col$opvertex-fiber gph$two))

(4) (SET.FTN$function opfirst)
    (= (SET.FTN$source opfirst) cocone)
    (= (SET.FTN$target opfirst) set.mor$morphism)
    (= (SET.FTN$composition [opfirst set.mor$source])
       (SET.FTN$composition [cocone-diagram set.dgm.pr$set1]))
    (= (SET.FTN$composition [opfirst set.mor$target]) opvertex)
    (forall (?s (cocone ?s))
        (= (opfirst ?s) (((set.col$component-fiber gph$two) ?s) 1)))

(5) (SET.FTN$function opsecond)
    (= (SET.FTN$source opsecond) cocone)
    (= (SET.FTN$target opsecond) set.mor$morphism)
    (= (SET.FTN$composition [opsecond set.mor$source])
       (SET.FTN$composition [cocone-diagram set.dgm.pr$set2]))
    (= (SET.FTN$composition [opsecond set.mor$target]) opvertex)
    (forall (?s (cocone ?s))
        (= (opsecond ?s) (((set.col$component-fiber gph$two) ?s) 2)))
```

There is a *colimiting cocone* or *choice* class function, which maps a pair (of sets) to its colimiting binary coproduct cocone. The totality of this function, along with the universality of the comediator set function, implies that a binary coproduct exists for any pair of sets. The opvertex of the colimiting binary coproduct cocone is a specific, but unidentified, *binary coproduct* set. It comes equipped with two *injection* functions. The binary coproduct and injections are expressed abstractly by their defining axioms. A binary coproduct colimiting cocone is the very special case of a colimiting cocone over a pair of sets. We connect this specific terminology to the generic fiber terminology for set colimits.

```
(6) (SET.FTN$function colimiting-cocone)
    (= (SET.FTN$source colimiting-cocone) set.dgm.pr$diagram)
    (= (SET.FTN$target colimiting-cocone) cocone)
    (= (SET.FTN$composition [colimiting-cocone cocone-diagram])
       (SET.FTN$identity set.dgm.pr$diagram))
    (= colimiting-cocone (set.col$colimiting-cocone-fiber gph$two))

(7) (SET.FTN$function colimit)
    (SET.FTN$function binary-coproduct)
    (= binary-coproduct colimit)
    (= (SET.FTN$source colimit) set.dgm.pr$diagram)
    (= (SET.FTN$target colimit) set.obj$object)
    (= colimit (SET.FTN$composition [colimiting-cocone opvertex]))
    (= colimit (set.col$colimit-fiber gph$two))

(8) (SET.FTN$function injection1)
    (= (SET.FTN$source injection1) set.dgm.pr$diagram)
    (= (SET.FTN$target injection1) set.mor$morphism)
    (= (SET.FTN$composition [injection1 set.mor$source]) set.dgm.pr$set1)
    (= (SET.FTN$composition [injection1 set.mor$target]) colimit)
    (= injection1 (SET.FTN$composition [colimiting-cocone opfirst]))
    (forall (?d (set.dgm.pr$diagram ?d))
        (= (injection1 ?d) (((set.col$injection-fiber gph$two) ?d) 1)))

(9) (SET.FTN$function injection2)
    (= (SET.FTN$source injection2) set.dgm.pr$diagram)
    (= (SET.FTN$target injection2) set.mor$morphism)
    (= (SET.FTN$composition [injection2 set.mor$source]) set.dgm.pr$set2)
    (= (SET.FTN$composition [injection2 set.mor$target]) colimit)
    (= injection2 (SET.FTN$composition [colimiting-cocone opsecond]))
    (forall (?d (set.dgm.pr$diagram ?d))
        (= (injection2 ?d) (((set.col$injection-fiber gph$two) ?d) 2)))
```

For the convenience of the implementer, we also provide a standard binary coproduct and a standard pair of injections functions that she may choose. The *standard coproduct* or *disjoint union* of a pair of sets

$$\mathit{std\text{-}coprd}(\boldsymbol{A}) = \boldsymbol{A}(1) \oplus \boldsymbol{A}(2)$$

provides a concrete binary coproduct for that pair. For any pair $\boldsymbol{A}$ there are *standard injection* functions

$$\mathit{std\text{-}inj}(\boldsymbol{A})(1) : \boldsymbol{A}(1) \to \boldsymbol{A}(1) \oplus \boldsymbol{A}(2) \text{ and } \mathit{std\text{-}inj}(\boldsymbol{A})(2) : \boldsymbol{A}(2) \to \boldsymbol{A}(1) \oplus \boldsymbol{A}(2)$$

defined by $\mathit{std\text{-}inj}(\boldsymbol{A})(1)(x) = (1, x)$ and $\mathit{std\text{-}inj}(\boldsymbol{A})(2)(x) = (2, x)$. Obviously, the standard injections are injective. The *standard colimiting cocone* is then constructed out of the standard binary coproduct and the two standard injection functions. Note how this proceeds bottom-up and in reverse order to the specific components.

```
(10) (SET.FTN$function standard-colimit)
     (SET.FTN$function standard-binary-coproduct)
     (SET.FTN$function disjoint-union)
     (= standard-binary-coproduct standard-colimit)
     (= disjoint-union standard-binary-coproduct)
     (= (SET.FTN$source standard-colimit) set.dgm.pr$diagram)
     (= (SET.FTN$target standard-colimit) set.obj$object)
     (forall (?p (set.dgm.pr$diagram ?p))
         (<=> ((standard-colimit ?p) ?z)
              (or (exists (?x ((set.dgm.pr$set1 ?p) ?x))
                      (= ?z [1 ?x]))
                  (exists (?x ((set.dgm.pr$set2 ?p) ?x))
                      (= ?z [2 ?x]))))))

(11) (KIF$function standard-injection1)
```

```
     (= (KIF$source standard-injection1) set.dgm.pr$diagram)
     (= (KIF$target standard-injection1) set.mor$morphism)
     (forall (?p (set.dgm.pr$diagram ?p)
              ?x1 ((set.dgm.pr$set1 ?p) ?x1))
         (= ((standard-injection1 ?p) ?x1) [1 ?x1]))

(12) (KIF$function standard-injection2)
     (= (KIF$source standard-injection2) set.dgm.pr$diagram)
     (= (KIF$target standard-injection2) set.mor$morphism)
     (forall (?p (set.dgm.pr$diagram ?p)
              ?x2 ((set.dgm.pr$set2 ?p) ?x2))
         (= ((standard-injection2 ?p) ?x2) [2 ?x2]))

(13) (SET.FTN$function standard-colimiting-cocone)
     (= (SET.FTN$source standard-colimiting-cocone) set.dgm.pr$diagram)
     (= (SET.FTN$target standard-colimiting-cocone) cocone)
     (= (SET.FTN$composition [standard-colimiting-cocone cocone-diagram])
        (SET.FTN$identity set.dgm.pr$diagram))
     (= (SET.FTN$composition [standard-colimiting-cocone opvertex]) standard-binary-coproduct)
     (= (SET.FTN$composition [standard-colimiting-cocone opfirst]) standard-injection1)
     (= (SET.FTN$composition [standard-colimiting-cocone opsecond]) standard-injection2)
```

For any binary coproduct cocone, there is a *comediator* set function $[f_1, f_2] : A_1 + A_2 \to A$ (see Figure 36, where arrows denote set functions) from the binary coproduct of the underlying diagram (pair of sets) to the opvertex of the cocone. This is the unique function, which commutes with the opfirst f_1 and opsecond f_2 functions. Existence and uniqueness represents the universality of the binary coproduct operator. Universality means that the comediator is the unique function that makes the diagram in Figure 36 commutative. We axiomatize this with a definite description. A binary coproduct comediator is the very special case of a comediator over a pair of sets. We connect this specific terminology to the generic fiber terminology for set colimits.

```
(14) (SET.FTN$function comediator)
     (= (SET.FTN$source comediator) cocone)
     (= (SET.FTN$target comediator) set.mor$morphism)
     (= (SET.FTN$composition [comediator set.mor$source])
        (SET.FTN$composition [cocone-diagram colimit]))
     (= (SET.FTN$composition [comediator set.mor$target]) opvertex)
     (forall (?s (cocone ?s))
         (= (comediator ?s)
            (the (?f (set.mor$morphism ?f))
                (and (= (set.mor$composition [(injection1 (cocone-diagram ?s)) ?f])
                        (opfirst ?s))
                     (= (set.mor$composition [(injection2 (cocone-diagram ?s)) ?f])
                        (opsecond ?s))))))
     (= comediator (set.col$comediator-fiber gph$two))
```

Binary Coproduct Morphisms

`set.col.coprd2.mor`

Binary coproducts are connected by morphisms (Figure 37). Morphisms offer a change of perspective for binary coproducts. The binary coproduct morphism maps a set pair morphism to a set function. As such, it is the morphism class function of a binary coproduct functor $+ : \mathsf{Set}^2 \to \mathsf{Set}$ from the functor category Set^2 to Set. The morphism of binary coproducts is defined as the comediator of a morphism cocone.

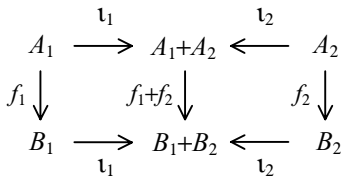


Figure 37: Binary Coproduct Morphism

```
(1) (SET.FTN$function morphism-cocone)
    (= (SET.FTN$source morphism-cocone) set.dgm.pr.mor$morphism)
    (= (SET.FTN$target morphism-cocone) set.col.coprd2$cocone)
    (= (SET.FTN$composition [morphism-cocone set.col.coprd2$cocone-diagram])
       set.dgm.pr.mor$source)
    (= (SET.FTN$composition [morphism-cocone set.col.coprd2$opvertex])
       (SET.FTN$composition [set.dgm.pr.mor$target colimit]))
    (forall (?f (set.dgm.pr.mor$morphism ?f))
        (and (= (set.col.coprd2$opfirst (morphism-cocone ?f))
                (set.ftn$composition
                    [(set.dgm.pr.mor$function1 ?f)
                     (set.col.coprd2$injection1 (set.dgm.pr.mor$target ?f))]))
             (= (set.col.coprd2$opsecond (morphism-cocone ?f))
                (set.ftn$composition
                    [(set.dgm.pr.mor$function2 ?f)
                     (set.col.coprd2$injection2 (set.dgm.pr.mor$target ?f))])))))

(2) (SET.FTN$function morphism)
    (= (SET.FTN$source morphism) set.dgm.pr.mor$morphism)
    (= (SET.FTN$target morphism) set.mor$morphism)
    (= (SET.FTN$composition [morphism set.mor$source])
       (SET.FTN$composition [set.dgm.pr.mor$source colimit]))
    (= (SET.FTN$composition [morphism set.mor$target])
       (SET.FTN$composition [set.dgm.pr.mor$target colimit]))
    (= morphism (SET.FTN$composition [morphism-cocone comediator]))
```

Endorelations and Quotients

set.col.endo

This section is crucial.

Endorelations are definable on the category Set. These are comparable to the endorelations in the namespace of relations. Let ***endo*** denote the class of all set endorelations. A set *endorelation* is a pair $\boldsymbol{J} = \langle A, E \rangle = \langle \mathit{set}(\boldsymbol{J}), \mathit{elem}(\boldsymbol{J}) \rangle$ consisting of a set $\mathit{set}(\boldsymbol{J}) = A$, and a binary endorelation $\mathit{elem}(\boldsymbol{J}) = E \subseteq A \times A$ on elements of A. The set A is called the *base set* of $\boldsymbol{J}$ – the set on which $\boldsymbol{J}$ is based. A set endorelation is determined by the pair consisting of its base set and its endorelation.

```
(1) (SET$class endorelation)

(2) (SET.FTN$function set)
    (SET.FTN$function base)
    (= base set)
    (= (SET.FTN$source set) endorelation)
    (= (SET.FTN$target set) set.obj$object)

(3) (SET.FTN$function element)
    (= (SET.FTN$source element) endorelation)
    (= (SET.FTN$target element) rel.endo$endorelation)
    (= (SET.FTN$composition [element rel.endo$set]) set)
```

The class ***endo*** of endorelations is ordered. For any two endorelations $\boldsymbol{J}_1, \boldsymbol{J}_2 \in \mathit{endo}$, $\boldsymbol{J}_1$ is a subrelation of $\boldsymbol{J}_2$, $\boldsymbol{J}_1 \leq \boldsymbol{J}_2$, when $\mathit{set}(\boldsymbol{J}_1) = \mathit{set}(\boldsymbol{J}_2)$ and $\mathit{elem}(\boldsymbol{J}_1) \subseteq \mathit{elem}(\boldsymbol{J}_2)$.

```
(4) (ORD$partial-order subrelation)
    (= (ORD$class subrelation) endorelation)
    (forall (?j1 (endorelation ?j1)
             ?j2 (endorelation ?j2))
        (<=> (subrelation ?j1 ?j2)
             (and (= (set ?j1) (set ?j2))
                  (rel.endo$subrelation (element ?j1) (element ?j2)))))
```

The symbol $\equiv_{\boldsymbol{E}}$ is the equivalence relation generated by (reflexive, symmetric and transitive closure of) $\mathit{elem}(\boldsymbol{J}) = E$. A simple inductive proof shows that the pair $\hat{\boldsymbol{J}} = \langle A, \equiv_{\boldsymbol{E}} \rangle$ is also an endorelation. Often the endorelation E is itself an equivalence relation, so that $\equiv_{\boldsymbol{E}}$ is E. However, it is not only convenient but also very important not to require this. In particular, the endorelations defined for computing the coequalizer of parallel pairs of set functions (coequalizer diagrams) are not equivalence relations.

Any set function $g : A \to B$ defines a set endorelation $\boldsymbol{J}_{\boldsymbol{g}} = \langle A, \boldsymbol{E}_{\boldsymbol{g}} \rangle$ called the *kernel* of g, where the base set is the source set A and the element endorelation is the kernel of the function: $\mathit{elem}(\boldsymbol{J}_{\boldsymbol{g}}) = \boldsymbol{E}_{\boldsymbol{g}} = \{(a_1, a_2) \mid g(a_1) = g(a_2)\}$.

```
(5) (SET.FTN$function kernel)
    (= (SET.FTN$source kernel) set.mor$morphism)
    (= (SET.FTN$target kernel) endorelation)
    (forall (?g (set.mor$morphism ?g))
        (and (= (set (kernel ?g)) (set.mor$source ?g))
             (= (element (kernel ?g)) (set.mor$kernel ?g))))
```

Let $\boldsymbol{J} = \langle A, E \rangle = \langle \mathit{set}(\boldsymbol{J}), \mathit{elem}(\boldsymbol{J}) \rangle$ be a set endorelation. A set function $g : A \to B$ *respects* $\boldsymbol{J}$ when $\boldsymbol{J}$ is a subrelation of the kernel of g, $\boldsymbol{J} \leq \boldsymbol{J}_{\boldsymbol{g}}$; that is, if $(a_0, a_1) \in \mathit{elem}(\boldsymbol{J})$ then $g(a_0) = g(a_1)$ for any two elements $a_0, a_1 \in A$. An easy induction shows that when g respects $\boldsymbol{J}$ it must also respect the equivalence closure $\hat{\boldsymbol{J}} = \langle A, \equiv_{\boldsymbol{E}} \rangle$. By definition, any set function respects its kernel (see below). Moreover, the kernel of any set function g is the "largest" set endorelation that g respects.

```
(6) (SET.LIM.PBK$opspan matches-opspan)
    (= (SET.LIM.PBK$class1 matches-opspan) set.mor$morphism)
    (= (SET.LIM.PBK$class2 matches-opspan) endorelation)
    (= (SET.LIM.PBK$opvertex matches-opspan) set.obj$object)
    (= (SET.LIM.PBK$opfirst matches-opspan) set.mor$source)
    (= (SET.LIM.PBK$opsecond matches-opspan) set)

(7) (REL$relation matches)
```

```
     (= (REL$class1 matches) set.mor$morphism)
     (= (REL$class2 matches) endorelation)
     (= (REL$extent matches) (SET.LIM.PBK$pullback matches-opspan))

(8) (REL$relation respects)
    (= (REL$class1 respects) set.mor$morphism)
    (= (REL$class2 respects) endorelation)
    (REL$subrelation respects matches)
    (forall (?f (set.mor$morphism ?f)
            ?j (endorelation ?j) (matches ?f ?j))
       (<=> (respects ?f ?j)
            (subrelation ?j (kernel ?f))))
```

Let $\boldsymbol{J} = \langle A, E \rangle = \langle \mathit{set}(\boldsymbol{J}), \mathit{elem}(\boldsymbol{J}) \rangle$ be a set endorelation.

A *quotient object* of a set A is a set B plus a surjection $e : A \to B$ whose source is A. For any element $a \in A$, the E-equivalence class of a is the set $[a]_{\boldsymbol{E}} = \{b \in A \mid b \equiv_{\boldsymbol{E}} a\}$. The set $A/\equiv_{\boldsymbol{E}}$ consisting of all equivalence classes and the *surjection* $[\text{-}]_{\boldsymbol{E}} : A \to A/\equiv_{\boldsymbol{E}}$, which maps any element $a \in A$ to its own equivalence class, is a *canonical quotient object*. This canonical surjection respects $\boldsymbol{J}$; in fact, the kernel of the canonical surjection is the equivalence closure $\hat{\boldsymbol{J}} = \langle A, \equiv_{\boldsymbol{E}} \rangle$. Also, if any function $g : A \to B$ respects $\boldsymbol{J}$, then $\hat{\boldsymbol{J}}$ is a subrelation of the kernel of g, $\hat{\boldsymbol{J}} \leq \boldsymbol{J}_g$. In summary, the kernel of the canonical surjection $[\text{-}]_{\boldsymbol{E}}$ is the smallest set endorelation containing $\boldsymbol{J}$. This expresses a universal property of the canonical surjection.

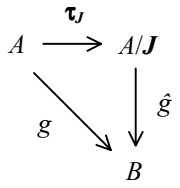


Diagram 35: Canonical Morphism and Universality of the Quotient

We abstract this universal property of the canonical surjection as a universality condition, which is stated in the definitions and proposition below. The universality for colimits such as coequalizers and pushouts in the category Set are expressed in the IFF in terms of this universality condition. The canonical surjection $[\text{-}]_{\boldsymbol{E}} : A \to A/\equiv_{\boldsymbol{E}}$ satisfies this universality condition. However, this is just one choice of many, all of which are isomorphic to each other.

The *canon*ical surjection $\boldsymbol{\tau}_{\boldsymbol{J}} : A \to A/\boldsymbol{J}$ of an endorelation $\boldsymbol{J} = \langle A, E \rangle = \langle \mathit{set}(\boldsymbol{J}), \mathit{elem}(\boldsymbol{J}) \rangle$ (see Diagram 35, where arrows denote set functions) is a chosen surjection that satisfies this universality condition stated in the proposition below. The *quotient* $A/\boldsymbol{J}$ is the target set of the canonical surjection. This canon-quotient pair is called the *canonical quotient object*. The totality of the function that chooses the canonical quotient object, along with the universality of the comediator function, axiomatizes the universality condition.

For the convenience of the implementer, for any set endorelation $\boldsymbol{J} = \langle A, E \rangle = \langle \mathit{set}(\boldsymbol{J}), \mathit{elem}(\boldsymbol{J}) \rangle$ we also provide the *standard quotient*

$\mathit{std\text{-}quo}(\boldsymbol{J}) = A/\equiv_{\boldsymbol{E}}$

and the *standard canon*

$\mathit{std\text{-}cnn}(\boldsymbol{J}) = [\text{-}]_{\boldsymbol{E}} : A \to \mathit{std\text{-}quo}(\boldsymbol{J})$

Obviously, the standard canon is a surjection.

```
 (9) (SET.FTN$function canon)
     (= (SET.FTN$source canon) endorelation)
     (= (SET.FTN$target canon) set.mor$surjection)
     (= (SET.FTN$composition [canon set.mor$source]) set)

(10) (SET.FTN$function quotient)
     (= (SET.FTN$source quotient) endorelation)
     (= (SET.FTN$target quotient) set.obj$object)
     (= quotient (SET.FTN$composition [canon set.mor$target]))

(11) (SET.FTN$function standard-quotient)
     (= (SET.FTN$source standard-quotient) endorelation)
     (= (SET.FTN$target standard-quotient) set.obj$object)
     (= standard-quotient
        (SET.FTN$composition
            [element
             (SET.FTN$composition [rel.endo$equivalence-closure rel.endo$quotient])]))
```

```
(12) (SET.FTN$function standard-canon)
     (= (SET.FTN$source standard-canon) endorelation)
     (= (SET.FTN$target standard-canon) set.mor$surjection)
     (= (SET.FTN$composition [standard-canon set.mor$source]) set)
     (= standard-quotient
        (SET.FTN$composition
            [element
             (SET.FTN$composition [rel.endo$equivalence-closure rel.endo$canon])]))
```

Proposition. *Let* $\boldsymbol{J}$ *be any set endorelation. The canonical surjection* $\boldsymbol{\tau_J} : A \to A/\boldsymbol{J}$ *respects* $\boldsymbol{J}$*. Also, any function* $g : A \to B$ *that respects* $\boldsymbol{J}$ *factors uniquely through the canonical quotient object; that is, there is a unique comediating function* $\hat{g} : A/\boldsymbol{J} \to B$ *such that* $\boldsymbol{\tau_J} \circ \hat{g} = g$.

This can be stated in terms of the order of set endorelations: the kernel of $\boldsymbol{\tau_J}$ is the smallest set endorelation containing $\boldsymbol{J}$. In more detail, $\boldsymbol{J}$ is a subrelation of the kernel of $\boldsymbol{\tau_J}$, $\boldsymbol{J} \leq \boldsymbol{J_{\tau J}}$; and for any set function $g : A \to B$, if $\boldsymbol{J}$ is a subrelation of the kernel of g, $\boldsymbol{J} \leq \boldsymbol{J_g}$, then the kernel of $\boldsymbol{\tau_J}$ is a subrelation of the kernel of g, $\boldsymbol{J_{\tau J}} \leq \boldsymbol{J_g}$. Since the canonical surjection $\boldsymbol{\tau_J} : A \to A/\boldsymbol{J}$ respects $\boldsymbol{J}$, its unique comediator must be the identity. Based on this proposition, a definite description is used to define a *comediator* function (Diagram 1) that maps a pair (g, $\boldsymbol{J}$) consisting of a set endorelation and a respectful set function to their comediator $\hat{g}$.

```
(13) (SET.FTN$function comediator)
     (= (SET.FTN$source comediator) (REL$extent respects))
     (= (SET.FTN$target comediator) set.mor$morphism)
     (forall (?g (set.mor$morphism ?g)
              ?j (endorelation ?j)
              (respects ?g ?j))
         (= (comediator [?g ?j])
            (the (?gt (set.mor$morphism ?gt))
                 (and (= (set.mor$source ?gt) (quotient ?j))
                      (= (set.mor$target ?gt) (set.mor$target ?g))
                      (= (set.mor$composition [(canon ?j) ?gt]) ?g))))))
```

From remarks above, the kernel of the canonical surjection $\boldsymbol{\tau_J} : A \to A/\boldsymbol{J}$ is the equivalence closure $\hat{\boldsymbol{J}} = \langle A, \equiv_{\boldsymbol{E}} \rangle$, the kernel of the canonical surjection $[-]_{\boldsymbol{E}} : A \to A/\equiv_{\boldsymbol{E}}$ is also the equivalence closure $\hat{\boldsymbol{J}} = \langle A, \equiv_{\boldsymbol{E}} \rangle$, and there is a bijection $A/\boldsymbol{J} \cong A/\equiv_{\boldsymbol{E}}$ making $\boldsymbol{\tau_J}$ and $[-]_{\boldsymbol{E}}$ isomorphic.

If the canonical surjection $[-]_{\boldsymbol{E}} : A \to A/\equiv_{\boldsymbol{E}}$ were chosen as the canon for $\boldsymbol{J}$, the comediator must satisfy $[]_{\boldsymbol{E}} \cdot \hat{g} = g$, and hence is definable by $\hat{g}([a]_{\boldsymbol{E}}) = g(a)$ for all elements $a \in A$. Hence, the respectful constraints on g imply that $\hat{g}$ is well defined. But we reiterate that the chosen quotient object $A/\boldsymbol{J}$, canon $\boldsymbol{\tau_J} : \boldsymbol{G} \to A/\boldsymbol{J}$ and comediator are not required to be equal to this special case, but only isomorphic to it.

```
(14) (forall (?j (endorelation ?j))
         (and (set.mor$isomorphic (quotient ?j) (standard-quotient ?j))
              (exists (?h (set.mor$bijection ?h))
                  (= (set.mor$composition [(canon ?j) ?h]) (standard-canon ?j))))))
```

This notion of a set function kernel gives a canonical approach (Figure 38) for an epi-mono factorization system for set functions. Let ε_g denote the canonical surjection of the kernel of g, and let μ_g denote the unique mediating morphism such that $\varepsilon_g \circ \mu_g = g$. This is an "image factorization" of g. If g respects a set endorelation $\boldsymbol{J}$, then there is a unique function $g_{\boldsymbol{J}} : A/\boldsymbol{J} \to A/\boldsymbol{J_g}$ such that $\boldsymbol{\tau_J} \circ g_{\boldsymbol{J}} = \varepsilon_{\boldsymbol{g}}$ and $g_{\boldsymbol{J}} \circ \mu_{\boldsymbol{g}} = \hat{g}$.

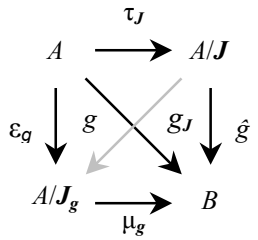


Figure 38: Factorization

Coequalizers

set.col.coeq

A *coequalizer* is a finite colimit in the category Set for a diagram of shape *parallel-pair* = • ⇉ •. Such a diagram is called a *parallel pair* of set functions.

The notion of a *coequalizer cocone* is used to specify and axiomatize coequalizers. Each coequalizer cocone is situated over a coequalizer *diagram* (parallel pair of set functions). And each coequalizer cocone (see Figure 39, where arrows denote set functions) has an *opvertex* set C, and a function f whose source set is the destination set of the functions in the underlying cocone diagram (parallel-pair) and whose target set is the opvertex. Since Figure 39 is a commutative diagram, the composite function $f_1 \cdot f = f_2 \cdot f$ is not needed. A coequalizer cocone is the very special case of a cocone over a parallel pair of functions. We connect this specific terminology to the generic fiber terminology for set colimits.

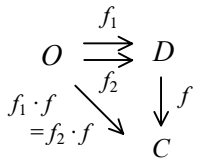


Figure 39: Coequalizer Cocone

```
(1) (SET$class cocone)
    (= cocone (set.col$cocone-fiber gph$parallel-pair)

(2) (SET.FTN$function cocone-diagram)
    (= (SET.FTN$source cocone-diagram) cocone)
    (= (SET.FTN$target cocone-diagram) set.dgm.ppr$diagram)
    (= cocone-diagram (set.col$cocone-tuple-fiber gph$parallel-pair))

(3) (SET.FTN$function opvertex)
    (= (SET.FTN$source opvertex) cocone)
    (= (SET.FTN$target opvertex) set.obj$object)
    (= opvertex (set.col$opvertex-fiber gph$parallel-pair))

(4) (SET.FTN$function function)
    (= (SET.FTN$source function) cocone)
    (= (SET.FTN$target function) set.mor$morphism)
    (= (SET.FTN$composition [function set.mor$source])
       (SET.FTN$composition [cocone-diagram set.dgm.ppr$destination]))
    (= (SET.FTN$composition [function set.mor$target]) opvertex)
    (forall (?s (cocone ?s))
        (= (function ?s) (((set.col$component-fiber gph$parallel-pair) ?s) 2)))

(5) (forall (?s (cocone ?s))
        (= (set.mor$composition [(set.dgm.ppr$function1 (cocone-diagram ?s)) (function ?s)])
           (set.mor$composition [(set.dgm.ppr$function2 (cocone-diagram ?s)) (function ?s)])))
```

It is important to observe that the set function in a cocone respects the endorelation of the underlying diagram of the cocone. This fact is needed to help define the comediator of a cocone, thereby expressing the universality property for coequalizers.

```
(6) (forall (?s (cocone ?s))
        (set.col.endo$respects (function ?s) (set.dgm.ppr$endorelation (cocone-diagram ?s))))
```

A *colimiting cocone* class function maps a parallel pair of set functions to its colimiting coequalizer cocone. See Figure 40, where arrows denote set functions. The totality of this function, along with the universality of the comediator set function, implies that a coequalizer exists for any parallel pair of set functions. The opvertex of the colimiting coequalizer cocone is a specifically chosen *coequalizer* set. It comes equipped with a *canon* set function. The coequalizer and canon are expressed both abstractly, and in the last axiom, specifically as the quotient and canon of the endorelation of the diagram. That is, the chosen universal objects for set endorelations and coequalizer diagrams (parallel pairs) are coordinated. A coequalizer colimiting cocone is the very special case of a colimiting cocone over a parallel

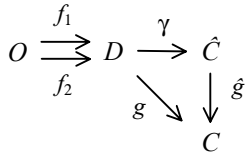


Figure 40: Colimiting Cocone, Coequalizer, and Comediator

pair of functions. We connect this specific terminology to the generic fiber terminology for set colimits. The fact that the canon is the "injection" fiber for the graph `parallel-pair`, indicates the abuse of terminology for colimit "injections".

```
(7) (SET.FTN$function colimiting-cocone)
    (= (SET.FTN$source colimiting-cocone) set.dgm.ppr$diagram)
    (= (SET.FTN$target colimiting-cocone) cocone)
    (= (SET.FTN$composition [colimiting-cocone cocone-diagram])
       (SET.FTN$identity set.dgm.ppr$diagram))
    (= colimiting-cocone (set.col$colimiting-cocone-fiber gph$parallel-pair))

(8) (SET.FTN$function colimit)
    (SET.FTN$function coequalizer)
    (= coequalizer colimit)
    (= (SET.FTN$source colimit) set.dgm.ppr$diagram)
    (= (SET.FTN$target colimit) set.obj$object)
    (= colimit (SET.FTN$composition [colimiting-cocone opvertex]))
    (= colimit (SET.FTN$composition [set.dgm.ppr$endorelation set.col.endo$quotient]))
    (= colimit (set.col$colimit-fiber gph$parallel-pair))

(9) (SET.FTN$function canon)
    (= (SET.FTN$source canon) set.dgm.ppr$diagram)
    (= (SET.FTN$target canon) set.mor$morphism)
    (= (SET.FTN$composition [canon set.mor$source]) set.dgm.ppr$destination])
    (= (SET.FTN$composition [canon set.mor$target]) colimit)
    (= canon (SET.FTN$composition [colimiting-cocone morphism]))
    (= canon (SET.FTN$composition [set.dgm.ppr$endorelation set.col.endo$canon]))
    (forall (?d (set.dgm.pr$diagram ?d))
        (= (canon ?d) (((set.col$injection-fiber gph$parallel-pair) ?d) 2)))
```

For the convenience of the implementer, for any parallel pair we also provide a *standard coequalizer* ***std-coeq***(***p***) that provides a concrete coequalizer for that parallel pair. In addition, there is a *standard canon*

std-cnn(***p***) : $D \to$ ***std-coeq***(***p***).

These are defined in terms of the standard quotient and standard canon of the endorelation of the pair. The *standard colimiting cocone* is then constructed out of the standard coequalizer and the standard canon functions. Note how this proceeds bottom-up and in reverse order to the specific components.

```
(10) (SET.FTN$function standard-colimit)
     (SET.FTN$function standard-coequalizer)
     (= standard-coequalizer standard-colimit)
     (= (SET.FTN$source standard-colimit) set.dgm.ppr$diagram)
     (= (SET.FTN$target standard-colimit) set.obj$object)
     (= standard-colimit
        (SET.FTN$composition
            [set.dgm.ppr$endorelation set.col.endo$standard-quotient]))

(11) (SET.FTN$function standard-canon)
     (= (SET.FTN$source standard-canon) set.dgm.ppr$diagram)
     (= (SET.FTN$target standard-canon) set.mor$morphism)
     (= (SET.FTN$composition [standard-canon set.mor$source]) set.dgm.ppr$destination])
     (= (SET.FTN$composition [standard-canon set.mor$target]) standard-colimit)
     (= standard-canon
        (SET.FTN$composition
            [set.dgm.ppr$endorelation set.col.endo$standard-canon]))

(12) (SET.FTN$function standard-colimiting-cocone)
     (= (SET.FTN$source standard-colimiting-cocone) set.dgm.ppr$diagram)
     (= (SET.FTN$target standard-colimiting-cocone) cocone)
     (= (SET.FTN$composition [standard-colimiting-cocone cocone-diagram])
        (SET.FTN$identity set.dgm.ppr$diagram))
     (= (SET.FTN$composition [standard-colimiting-cocone opvertex]) standard-coequalizer)
     (= (SET.FTN$composition [standard-colimiting-cocone opfirst]) standard-canon)
```

The *comediator* set function, from the coequalizer of the underlying diagram (parallel pair of set functions) to the opvertex of a cocone, is the unique set function that commutes with cocone set functions. See Figure 34, where arrows denote set functions. This is defined using a definite description, which expresses the universal property of the coequalizer. It is defined specifically as the comediator of the associated (morphism, set endorelation) pair. A coequalizer comediator is the very special case of a comediator over a

parallel pair of functions. We connect this specific terminology to the generic fiber terminology for set colimits.

```
(13) (SET.FTN$function comediator)
     (= (SET.FTN$source comediator) cocone)
     (= (SET.FTN$target comediator) set.mor$morphism)
     (= (SET.FTN$composition [comediator set.mor$source])
        (SET.FTN$composition [cocone-diagram colimit]))
     (= (SET.FTN$composition [comediator set.mor$target]) opvertex)
     (forall (?s (cocone ?s))
         (= (comediator ?s)
            (the (?m (set.mor$morphism ?m))
                 (= (set.mor$composition [(canon (cocone-diagram ?s)) ?m])
                    (morphism ?s)))))
     (= comediator (set.col$comediator-fiber gph$parallel-pair))

(14) (forall (?s (cocone ?s))
         (= (comediator ?s)
            (set.col.endo$comediator
                [(morphism ?s) (set.dgm.ppr$endorelation (cocone-diagram ?s))])))
```

Coequalizer Morphisms

`set.col.coeq.mor`

Coequalizers are connected by morphisms (Figure 41). Morphisms offer a change of perspective for coequalizers. The coequalizer morphism maps a set parallel pair morphism to a set function. As such, it is the morphism class function of a coequalizer functor $coeq : \mathsf{Set}^{\rightrightarrows} \to \mathsf{Set}$ from the functor category $\mathsf{Set}^{\rightrightarrows}$ to Set. The morphism of coequalizers is defined as the comediator of a morphism cocone.

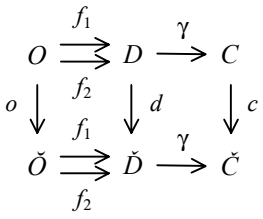


Figure 41: Coequalizer Morphism

```
(1) (SET.FTN$function morphism-cocone)
    (= (SET.FTN$source morphism-cocone) set.dgm.ppr.mor$morphism)
    (= (SET.FTN$target morphism-cocone) set.col.coeq$cocone)
    (= (SET.FTN$composition [morphism-cocone set.col.coeq$cocone-diagram])
       set.dgm.ppr.mor$source)
    (= (SET.FTN$composition [morphism-cocone set.col.coeq$opvertex])
       (SET.FTN$composition [set.dgm.ppr.mor$target set.col.coeq$colimit]))
    (forall (?f (set.dgm.ppr.mor$morphism ?f))
        (= (set.col.coeq$function (morphism-cocone ?f))
           (set.ftn$composition
               [(set.dgm.ppr.mor$destination ?f)
                (set.col.coeq$canon (set.dgm.ppr.mor$target ?f))])))

(2) (SET.FTN$function morphism)
    (= (SET.FTN$source morphism) set.dgm.ppr.mor$morphism)
    (= (SET.FTN$target morphism) set.mor$morphism)
    (= (SET.FTN$composition [morphism set.mor$source])
       (SET.FTN$composition [set.dgm.ppr.mor$source colimit]))
    (= (SET.FTN$composition [morphism set.mor$target])
       (SET.FTN$composition [set.dgm.ppr.mor$target colimit]))
    (= morphism (SET.FTN$composition [morphism-cocone comediator]))
```

Pushouts

set.col.psh

A *pushout* (Figure 42) is a finite colimit in the category Set for a diagram of shape *span* = • ← • → •. Such a diagram (of sets and set functions) is called a *span*.

Pushout cocones are used to specify and axiomatize pushouts. Each pushout cocone (Figure 42, where arrows denote set functions) has an underlying *diagram* (the shaded part of Figure 42), an *opvertex* set *A*, and a pair of set functions called *opfirst* and *opsecond*. Their common target set is the opvertex and their source sets are the target sets of the functions in the span. The opfirst and opsecond functions form a commutative diagram with the span. The term 'cocone' denotes the *pushout cocone* class. The term 'cocone-diagram' represents the underlying diagram. A pushout cocone is the very special case of a cocone over a span. We connect this specific terminology to the generic fiber terminology for set colimits.

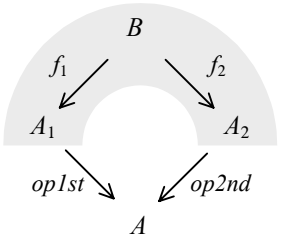


Figure 42: Pushout Cocone

```
(1) (SET$class cocone)
    (= cocone (set.col$cocone-fiber gph$span)

(2) (SET.FTN$function cocone-diagram)
    (= (SET.FTN$source cocone-diagram) cocone)
    (= (SET.FTN$target cocone-diagram) set.dgm.spn$diagram)
    (= cocone-diagram (set.col$cocone-tuple-fiber gph$span))

(3) (SET.FTN$function opvertex)
    (= (SET.FTN$source opvertex) cocone)
    (= (SET.FTN$target opvertex) set.obj$object)
    (= opvertex (set.col$opvertex-fiber gph$span))

(4) (SET.FTN$function opfirst)
    (= (SET.FTN$source opfirst) cocone)
    (= (SET.FTN$target opfirst) set.mor$morphism)
    (= (SET.FTN$composition [opfirst set.mor$source])
       (SET.FTN$composition [cocone-diagram set.dgm.spn$set1]))
    (= (SET.FTN$composition [opfirst set.mor$target]) opvertex)
    (forall (?s (cocone ?s))
        (= (opfirst ?s) (((set.col$component-fiber gph$span) ?s) 1)))

(5) (SET.FTN$function opsecond)
    (= (SET.FTN$source opsecond) cocone)
    (= (SET.FTN$target opsecond) set.mor$morphism)
    (= (SET.FTN$composition [opsecond set.mor$source])
       (SET.FTN$composition [cocone-diagram set.dgm.spn$set2]))
    (= (SET.FTN$composition [opsecond set.mor$target]) opvertex)
    (forall (?s (cocone ?s))
        (= (opsecond ?s) (((set.col$component-fiber gph$span) ?s) 2)))

(6) (forall (?s (cocone ?s))
        (= (set.mor$composition [(set.dgm.spn$first (cocone-diagram ?s)) (opfirst ?s)])
           (set.mor$composition [(set.dgm.spn$second (cocone-diagram ?s)) (opsecond ?s)])))
```

The *binary-coproduct cocone* underlying any cocone (pushout diagram) is named.

```
(7) (SET.FTN$function binary-coproduct-cocone)
    (= (SET.FTN$source binary-coproduct-cocone) cocone)
    (= (SET.FTN$target binary-coproduct-cocone) set.col.coprd2$cocone)
    (= (SET.FTN$composition [binary-coproduct-cocone set.col.coprd2$cocone-diagram])
       (SET.FTN$composition [cocone-diagram set.dgm.spn$pair]))
    (= (SET.FTN$composition [binary-coproduct-cocone set.col.coprd2$opvertex]) opvertex)
    (= (SET.FTN$composition [binary-coproduct-cocone set.col.coprd2$opfirst]) opfirst)
    (= (SET.FTN$composition [binary-coproduct-cocone set.col.coprd2$opsecond]) opsecond)
```

The *coequalizer cocone* function maps a pushout cocone of set functions to the associated (`set.col.coeq`) coequalizer cocone of set functions (see Figure 43, where arrows denote set functions). For this coequalizer cocone, the underlying coequalizer diagram is the coequalizer diagram associated with the underlying pushout diagram, the opvertex is the opvertex of the pushout cocone, and the set function is the binary coproduct comediator of the binary coproduct cocone (opfirst and opsecond functions with respect to the pair diagram of the cocone). This is the first step in the definition of the specific pushout of a pushout cocone. The following string of equalities demonstrates that this cocone is well defined

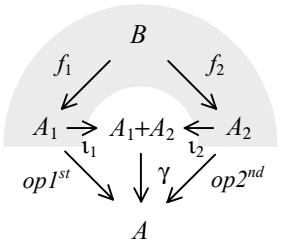


Figure 43: Coequalizer Cocone

$$f_1 \cdot \iota_1 \cdot \gamma = f_1 \cdot op1^{st} = f_2 \cdot op2^{nd} = f_2 \cdot \iota_2 \cdot \gamma.$$

```
(8) (SET.FTN$function coequalizer-cocone)
    (= (SET.FTN$source coequalizer-cocone) cocone)
    (= (SET.FTN$target coequalizer-cocone) set.col.coeq$cocone)
    (= (SET.FTN$composition [coequalizer-cocone set.col.coeq$cocone-diagram])
       (SET.FTN$composition [cocone-diagram set.dgm.spn$parallel-pair]))
    (= (SET.FTN$composition [coequalizer-cocone set.col.coeq$opvertex]) opvertex)
    (= (SET.FTN$composition [coequalizer-cocone set.col.coeq$function])
       (SET.FTN$composition [binary-coproduct-cocone set.col.coprd2$comediator]))
```

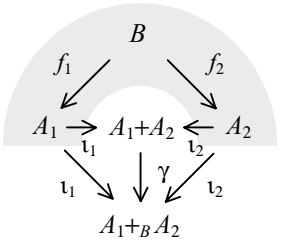


Figure 44: Colimiting Cocone

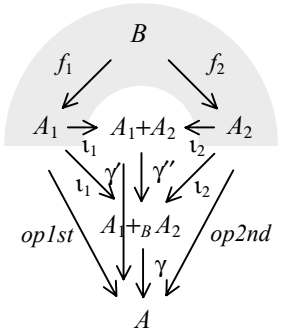


Figure 45: Comediator

The *colimiting cocone* class function maps a span to its colimiting pushout cocone. See Figure 44, where arrows denote set functions. For convenience of reference, we define three terms that represent the components of this pushout cocone. The opvertex of the pushout cocone is a specific *pushout* set, which comes equipped with two *injection* functions. The last axiom expresses the specificity 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. These axioms coordinate the choices for coproducts, coequalizers and pushouts. A pushout colimiting cocone is the very special case of a colimiting cocone over a span of sets and functions. We connect this specific terminology to the generic fiber terminology for set colimits.

```
 (9) (SET.FTN$function colimiting-cocone)
     (= (SET.FTN$source colimiting-cocone) set.dgm.spn$diagram)
     (= (SET.FTN$target colimiting-cocone) cocone)
     (= (SET.FTN$composition [colimiting-cocone cocone-diagram])
        (SET.FTN$identity set.dgm.spn$diagram))
     (= colimiting-cocone (set.col$colimiting-cocone-fiber gph$span))

(10) (SET.FTN$function colimit)
     (SET.FTN$function pushout)
     (= pushout colimit)
     (= (SET.FTN$source colimit) set.dgm.spn$diagram)
     (= (SET.FTN$target colimit) set.obj$object)
     (= colimit (SET.FTN$composition [colimiting-cocone opvertex]))
     (= colimit (set.col$colimit-fiber gph$span))
```

```
(11) (SET.FTN$function injection1)
     (= (SET.FTN$source injection1) set.dgm.spn$diagram)
     (= (SET.FTN$target injection1) set.mor$morphism)
     (= (SET.FTN$composition [injection1 set.mor$source]) set.dgm.spn$set1)
     (= (SET.FTN$composition [injection1 set.mor$target]) colimit)
     (= injection1 (SET.FTN$composition [colimiting-cocone opfirst])
     (forall (?d (set.dgm.pr$diagram ?d))
         (= (injection1 ?d) (((set.col$injection-fiber gph$span) ?d) 1)))

(12) (SET.FTN$function injection2)
     (= (SET.FTN$source injection2) set.dgm.spn$diagram)
     (= (SET.FTN$target injection2) set.mor$morphism)
     (= (SET.FTN$composition [injection2 set.mor$source]) set.dgm.spn$set2)
     (= (SET.FTN$composition [injection2 set.mor$target]) colimit)
     (= injection2 (SET.FTN$composition [colimiting-cocone opsecond])
     (forall (?d (set.dgm.pr$diagram ?d))
         (= (injection2 ?d) (((set.col$injection-fiber gph$span) ?d) 2)))

(13) (forall (?d (set.dgm.spn$diagram ?d))
         (and (= (colimit ?d)
                 (set.col.coeq$coequalizer (set.dgm.spn$parallel-pair ?d)))
              (= (injection1 ?d)
                 (set.mor$composition
                     [(set.col.coprd2$injection1 (set.dgm.spn$pair ?d))
                      (set.col.coeq$canon (set.dgm.spn$parallel-pair ?d))]))
              (= (injection2 ?d)
                 (set.mor$composition
                     [(set.col.coprd2$injection2 (set.dgm.spn$pair ?d))
                      (set.col.coeq$canon (set.dgm.spn$parallel-pair ?d))]))))
```

For the convenience of the implementer, for any span we also provide a *standard pushout* $\mathit{std\text{-}psh}(\boldsymbol{r}) = \boldsymbol{A}(1) \oplus_{\boldsymbol{B}} \boldsymbol{A}(2)$ that provides a concrete pushout for that span. In addition, there is a pair of *standard injection* functions

$\mathit{std\text{-}inj}(\boldsymbol{r})(1) : \boldsymbol{A}(1) \to \boldsymbol{A}(1) \oplus_{\boldsymbol{B}} \boldsymbol{A}(2)$ and $\mathit{std\text{-}inj}(\boldsymbol{r})(2) : \boldsymbol{A}(2) \to \boldsymbol{A}(1) \oplus_{\boldsymbol{B}} \boldsymbol{A}(2)$.

These are defined in terms of the standard coproduct of the pair of the span, and the standard coequalizer and standard canon of the standard parallel pair of the span. The *standard colimiting cocone* is then constructed out of the standard pushout and the standard injection functions. Note how this proceeds bottom-up and in reverse order to the specific components.

```
(14) (SET.FTN$function standard-colimit)
     (SET.FTN$function standard-pushout)
     (= standard-pushout standard-colimit)
     (= (SET.FTN$source standard-colimit) set.dgm.spn$diagram)
     (= (SET.FTN$target standard-colimit) set.obj$object)

(15) (SET.FTN$function standard-injection1)
     (= (SET.FTN$source standard-injection1) set.dgm.spn$diagram)
     (= (SET.FTN$target standard-injection1) set.mor$morphism)
     (= (SET.FTN$composition [standard-injection1 set.mor$source]) set.dgm.spn$set1)
     (= (SET.FTN$composition [standard-injection1 set.mor$target]) standard-colimit)

(16) (SET.FTN$function standard-injection2)
     (= (SET.FTN$source standard-injection2) set.dgm.spn$diagram)
     (= (SET.FTN$target standard-injection2) set.mor$morphism)
     (= (SET.FTN$composition [standard-injection2 set.mor$source]) set.dgm.spn$set2)
     (= (SET.FTN$composition [standard-injection2 set.mor$target]) standard-colimit)

(17) (forall (?d (set.dgm.spn$diagram ?d))
         (and (= (standard-colimit ?d)
                 (set.col.coeq$standard-coequalizer (set.dgm.spn$standard-parallel-pair ?d)))
              (= (standard-injection1 ?d)
                 (set.mor$composition
                     [(set.col.coprd2$standard-injection1 (set.dgm.spn$pair ?d))
                      (set.col.coeq$standard-canon (set.dgm.spn$standard-parallel-pair ?d))]))
              (= (injection2 ?d)
                 (set.mor$composition
                     [(set.col.coprd2$standard-injection2 (set.dgm.spn$pair ?d))
                      (set.col.coeq$standard-canon (set.dgm.spn$standard-parallel-pair ?d))]))))
```

```
(18) (SET.FTN$function standard-colimiting-cocone)
     (= (SET.FTN$source standard-colimiting-cocone) set.dgm.spn$diagram)
     (= (SET.FTN$target standard-colimiting-cocone) cocone)
     (= (SET.FTN$composition [standard-colimiting-cocone cocone-diagram])
        (SET.FTN$identity set.dgm.spn$diagram))
     (= (SET.FTN$composition [standard-colimiting-cocone opvertex]) standard-pushout)
     (= (SET.FTN$composition [standard-colimiting-cocone opfirst]) standard-injection1)
     (= (SET.FTN$composition [standard-colimiting-cocone opsecond]) standard-injection2)
```

For any pushout cocone, the *comediator* set function, from the pushout of the underlying diagram of the cocone to the opvertex of a cocone, is the unique set function that commutes with the opfirst and opsecond functions. See Figure 45, where arrows denote set functions. This is defined abstractly by using a definite description, and is defined specifically as the comediator of a coequalizer cocone. A pushout comediator is the very special case of a comediator over a span of sets and functions. We connect this specific terminology to the generic fiber terminology for set colimits.

```
(19) (SET.FTN$function comediator)
     (= (SET.FTN$source comediator) cocone)
     (= (SET.FTN$target comediator) set.mor$morphism)
     (= (SET.FTN$composition [comediator set.mor$source])
        (SET.FTN$composition [cocone-diagram colimit]))
     (= (SET.FTN$composition [comediator set.mor$target]) opvertex)
     (forall (?s (cocone ?s))
         (= (comediator ?s)
            (the (?m (set.mor$morphism ?m))
                 (and (= (composition [(injection1 (cocone-diagram ?s)) ?m])
                         (opfirst ?s))
                      (= (composition [(injection2 (cocone-diagram ?s)) ?m])
                         (opsecond ?s))))))
     (= comediator (set.col$comediator-fiber gph$span))

(20) (= comediator
        (SET.FTN$composition [coequalizer-cocone set.col.coeq$comediator]))
```

Pushout Morphisms

`set.col.psh.mor`

Pushouts are connected by morphisms (Figure 46). Morphisms offer a change of perspective for pushouts. The pushout morphism maps a set span to a set function. As such, it is the morphism class function of a pushout functor $psh : \mathsf{Set}^{\leftarrow\rightarrow} \to \mathsf{Set}$ from the functor category $\mathsf{Set}^{\leftarrow\rightarrow}$ to Set. The morphism of pushouts is defined as the comediator of a morphism cocone.

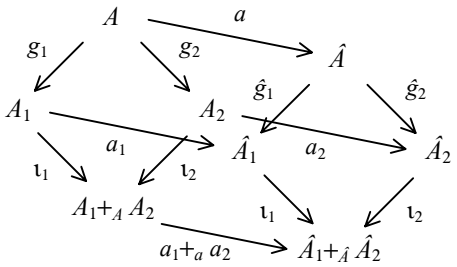


Figure 46: Pushout Morphism

```
(1) (SET.FTN$function morphism-cocone)
    (= (SET.FTN$source morphism-cocone) set.dgm.spn.mor$morphism)
    (= (SET.FTN$target morphism-cocone) set.col.psh$cocone)
    (= (SET.FTN$composition [morphism-cocone set.col.psh$cocone-diagram])
       set.dgm.spn.mor$source)
    (= (SET.FTN$composition [morphism-cocone set.col.psh$opvertex])
       (SET.FTN$composition [set.dgm.spn.mor$target set.col.psh$colimit]))
    (forall (?f (set.dgm.spn.mor$morphism ?f))
        (and (= (set.col.psh$opfirst (morphism-cocone ?f))
                (set.ftn$composition
                    [(set.dgm.spn.mor$function1 ?f)
                     (set.col.psh$injection1 (set.dgm.spn.mor$target ?f))]))
             (= (set.col.psh$opsecond (morphism-cocone ?f))
                (set.ftn$composition
                    [(set.dgm.spn.mor$function2 ?f)
                     (set.col.psh$injection2 (set.dgm.spn.mor$target ?f))])))))

(2) (SET.FTN$function morphism)
    (= (SET.FTN$source morphism) set.dgm.spn.mor$morphism)
    (= (SET.FTN$target morphism) set.mor$morphism)
    (= (SET.FTN$composition [morphism set.mor$source])
       (SET.FTN$composition [set.dgm.spn.mor$source colimit]))
    (= (SET.FTN$composition [morphism set.mor$target])
       (SET.FTN$composition [set.dgm.spn.mor$target colimit]))
    (= morphism (SET.FTN$composition [morphism-cocone comediator]))
```