The IFF Category Theory Ontology

This document declares and axiomatizes the IFF Category Theory Ontology. The IFF Category Theory Ontology provides a baseline formalism for category theory. It is anticipated that much more terminology will be added subsequently. As illustrated in Diagram 1, the collections and functions axiomatized in the top metalevel characterize the overall architecture for the large aspect of the Category Theory Ontology. Nodes in this diagram represent collections and arrows represent functions. The small oval on the right represents the namespace of large collections (classes) and their functions. The next larger oval represents the namespace of large graphs and their morphisms. These two namespaces are contained in the IFF Core Ontology. Also indicated are namespaces for categories, functors, natural transformations and adjunctions that are represented in the IFF Category Theory Ontology as defined in this document. Not illustrated are the namespaces for monads and colimits. The IFF Category Theory Ontology currently contains 224 non-identical terms (238 terms with 14 synonyms), partitioned into 43 terms for categories (`CAT`), 31 terms for functors (`FUNC`), 15 terms for natural transformations (`NAT`), 21 terms for adjunctions (`ADJ`, `ADJ.MOR`), 37 for monads (`MND`, `MND.EM`, `MND.KLI`), and 77 terms for colimits (`COL`, `COL.INIT`, `COL.COPRD2`, `COL.COEQ`, `COL.PSH`).

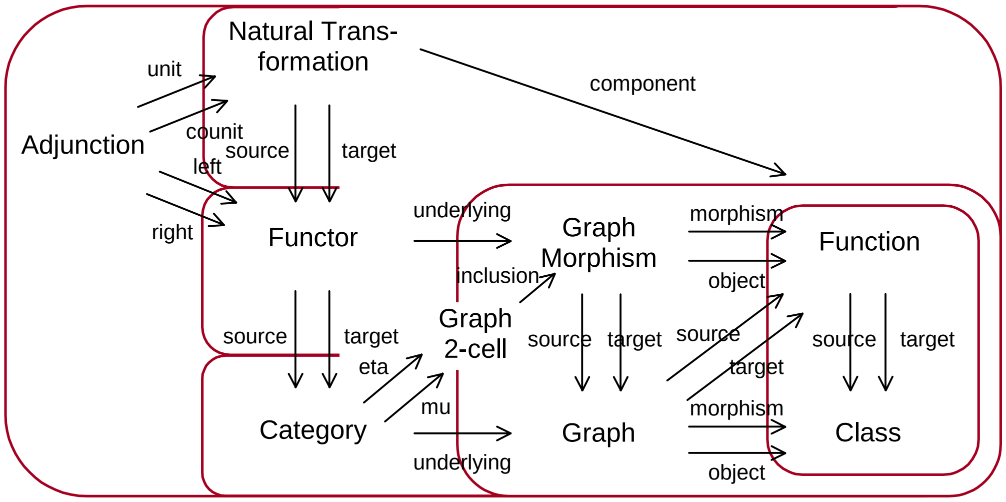


Diagram 1: Core Collections and Functions

The Namespace of Large Categories

`CAT`

This namespace represents large categories – categories whose collections of objects and morphisms are classes. The content of the namespace for categories is developed in chapter 1 of Mac Lane 1971. All terms declared and axiomatized in this namespace are listed in Table 1. The suggested prefix for this namespace is ‘`CAT`’, standing for category: when used in an external namespace, all terms that originate from this namespace should be prefixed with ‘`CAT`’.

Table 1: Terms introduced in the Category Theory Ontology

	Collection	Function	Other

`CAT`	`category`
	`small`
	`discrete`
		`graph = underlying mu eta`
		`object morphism`
		`source target`
		`composable-opspan composable`
		`first second`
		`composition identity`
		`opposition`
		`object-pair object-binary-product`
		`morphism-pair morphism-binary-product`
		`source-target object-hom morphism-hom`
		`left-composable left-composition`
		`right-composable right-composition`
		`epimorphism monomorphism isomorphism`
		`isomorphic`
			`subcategory`
			`empty`
			`two`
			`parallel-pair`
			`span`
			`terminal = unit = one`
			`Set`
			`Classification`

Basics

$$\begin{tikzcd} mor(|C_0|) \arrow[r, shift left, "src(|C_0|)"] \arrow[r, shift right, "tgt(|C_0|)"'] & obj(|C_0|) \end{tikzcd}$$

Figure 1: Category

A *category* $\boldsymbol{C}$ can be thought of as a special kind of graph $|\boldsymbol{C}|$ – a graph with monoidal properties (Figure 1). More precisely, a category $\boldsymbol{C} = \langle C, \mu_C, \eta_C \rangle$ is a monoid in the 2-dimensional quasi-category GRAPH of (large) graphs and graph morphisms. This means that it consists of an *underlying graph* $|\boldsymbol{C}|$, a composition graph 2-cell $\mu_C : |\boldsymbol{C}| \otimes |\boldsymbol{C}| \Rightarrow |\boldsymbol{C}|$ and an identity graph 2-cell $\eta_C : 1_{obj(C)} \Rightarrow |\boldsymbol{C}|$, both with the identity object function $id_{obj(C)}$. The 2-cell μ_C provides for a binary associative operation on morphisms in the category – its morphism function operates on any two morphisms that are *composable*, in the sense that the target object of the first is equal to the source object of the second, and returns a well-defined (composition) morphism. The 2-cell η_C

provides identities – its morphism function associates a well-defined (identity) morphism with each object in the category. Both mu and eta have the category as a parameter. Categories are determined by their (graph, mu, eta) triples.

```
(1) (KIF$collection category)

(2) (KIF$function graph)
    (KIF$function underlying)
    (= underlying graph)
    (= (KIF$source graph) category)
    (= (KIF$target graph) GPH$graph)

(3) (KIF$function mu)
    (= (KIF$source mu) category)
    (= (KIF$target mu) GPH.MOR$2-cell)
    (forall (?c (category ?c))
        (and (= (GPH.MOR$source (mu ?c))
                (GPH$multiplication [(graph ?c) (graph ?c)]))
             (= (GPH.MOR$target (mu ?c)) (graph ?c))))

(4) (KIF$function eta)
    (= (KIF$source eta) category)
    (= (KIF$target eta) GPH.MOR$2-cell)
    (forall (?c (category ?c))
        (and (= (GPH.MOR$source (eta ?c)) (GPH$unit (GPH$object (graph ?c))))
             (= (GPH.MOR$target (eta ?c)) (graph ?c))))

    (forall (?c1 (category ?c1) ?c2 (category ?c2))
        (=> (and (= (graph ?c1) (graph ?c2))
                 (= (mu ?c1) (mu ?c2))
                 (= (eta ?c1) (eta ?c2)))
            (= ?c1 ?c2)))
```

○ For convenience in the language used for categories, we rename the object and morphism classes, and the source and target functions in the setting of categories. These terms refer to the underlying graph of a category.

```
(5) (KIF$function object)
    (= (KIF$source object) category)
    (= (KIF$target object) SET$class)
    (forall (?c (category ?c))
        (= (object ?c) (GPH$object (graph ?c))))

(6) (KIF$function morphism)
    (= (KIF$source morphism) category)
    (= (KIF$target morphism) SET$class)
    (forall (?c (category ?c))
        (= (morphism ?c) (GPH$morphism (graph ?c))))

(7) (KIF$function source)
    (= (KIF$source source) category)
    (= (KIF$target source) SET.FTN$function)
    (forall (?c (category ?c))
        (= (source ?c) (GPH$source (graph ?c))))

(8) (KIF$function target)
    (= (KIF$source target) category)
    (= (KIF$target target) SET.FTN$function)
    (forall (?c (category ?c))
        (= (target ?c) (GPH$target (graph ?c))))
```

○ For convenience in the language used for categories, we define the relation of composable pairs of morphisms, and rename the first and second functions in the setting of categories. These terms refer to the horizontal multiplication of the underlying graph of a category with itself.

```
(9) (KIF$function composable-opspan)
    (= (KIF$source composable-opspan) category)
    (= (KIF$target composable-opspan) SET.LIM.PBK$opspan)
    (forall (?c (category ?c))
        (= (composable-opspan ?c)
           (GPH$multiplication-opspan [(graph ?c) (graph ?c)])))

(10) (KIF$function composable)
     (= (KIF$source composable) category)
     (= (KIF$target composable) REL$relation)
     (forall (?c (category ?c))
         (and (REL$class1 (composable ?c)) (morphism ?c))
              (REL$class2 (composable ?c)) (morphism ?c))
              (= (composable ?c)
                 (SET.LIM.PBK$relation (composable-opspan ?c)))))

(11) (KIF$function first)
     (= (KIF$source first) category)
     (= (KIF$target first) SET.FTN$function)
     (forall (?c (category ?c))
         (= (first ?c)
            (GPH$source (GPH$multiplication [(graph ?c) (graph ?c)]))))

(12) (KIF$function second)
     (= (KIF$source second) category)
     (= (KIF$target second) SET$function)
     (forall (?c (category ?c))
         (= (second ?c)
            (GPH$target (GPH$multiplication [(graph ?c)(graph ?c)]))))
```

- For convenience in the language used for categories, we rename the composition and identity operations in the setting of categories. These terms refer to the μ_C and η_C graph 2-cells of a category. Table 2 gives the notation for the *composition* function $\circ^C = mor(\mu_C)$ and the *identity* function $id^C = mor(\eta_C)$ – the morphism functions of the composition and identity graph morphisms.

$\circ^C : mor(\boldsymbol{C}) \times_{obj(\boldsymbol{C})} mor(\boldsymbol{C}) \rightarrow mor(\boldsymbol{C})$	$id^C : obj(\boldsymbol{C}) \rightarrow mor(\boldsymbol{C})$
$(m_1, m_2) \mapsto m_1 \circ m_2$	$o \mapsto id_o$

Table 2: Elements of Monoidal Structure

The composition function provides for a binary associative operation on morphisms in the category – it operates on any two morphisms that are *composable*, in the sense that the target object of the first is equal to the source object of the second, and returns a well-defined (composition) morphism. The identity function provides identities – it associates a well-defined (identity) morphism with each object in the category. Both composition and identity have the category as a parameter. Categories are determined by their (underlying, mu, eta) triples, and hence by their (underlying, composition, identity) triples.

```
(13) (KIF$function composition)
     (= (KIF$source composition) category)
     (= (KIF$target composition) SET.FTN$function)
     (forall (?c (category ?c))
         (= (composition ?c) (GPH.MOR$morphism (mu ?c))))

(14) (KIF$function identity)
     (= (KIF$source identity) category)
     (= (KIF$target identity) SET$function)
     (forall (?c (category ?c))
         (= (identity ?c) (GPH.MOR$morphism (eta ?c))))
```

- By the definitions of graph morphisms, graph multiplication and graph units, for any category $\boldsymbol{C}$ these operations satisfy the typing constraints listed in Table 3.

$\circ^C \cdot src(\boldsymbol{C}) = 1^{st}(\boldsymbol{C}) \cdot src(\boldsymbol{C})$

$\circ^{\boldsymbol{C}} \cdot tgt(\boldsymbol{C}) = 2^{nd}(\boldsymbol{C}) \cdot tgt(\boldsymbol{C})$
$id^{\boldsymbol{C}} \cdot src(\boldsymbol{C}) = id_{obj(\boldsymbol{C})} = id^{\boldsymbol{C}} \cdot tgt(\boldsymbol{C})$

Table 3: Preservation of Source and Target

- Table 4 contains commutative diagrams involving the μ and η graph 2-cells of categories and the coherence graph morphisms α, λ and ρ. The commutative diagram on the left represents the *associative law* for composition, and the commutative diagrams on the right represent the left and right *unit laws* for identity.

$\begin{tikzcd} \lvert\boldsymbol{C}\rvert \otimes (\lvert\boldsymbol{C}\rvert \otimes \lvert\boldsymbol{C}\rvert) \arrow[r, "\lvert\boldsymbol{C}\rvert \otimes \mu_{\lvert C\rvert}"] \arrow[d, "\alpha_{\lvert C\rvert, \lvert C\rvert, \lvert C\rvert}"'] & \lvert\boldsymbol{C}\rvert \otimes \lvert\boldsymbol{C}\rvert \arrow[dd, "\mu_{\lvert C\rvert}"] \\ (\lvert\boldsymbol{C}\rvert \otimes \lvert\boldsymbol{C}\rvert) \otimes \lvert\boldsymbol{C}\rvert \arrow[d, "\mu_{\lvert C\rvert} \otimes \lvert\boldsymbol{C}\rvert"'] & \\ \lvert\boldsymbol{C}\rvert \otimes \lvert\boldsymbol{C}\rvert \arrow[r, "\mu_{\lvert C\rvert}"'] & \lvert\boldsymbol{C}\rvert \end{tikzcd}$	$\begin{tikzcd} 1_C \otimes \lvert\boldsymbol{C}\rvert \arrow[r, "\eta_{\lvert C\rvert} \otimes \lvert\boldsymbol{C}\rvert"] \arrow[dr, "\lambda_{\lvert C\rvert}"'] & \lvert\boldsymbol{C}\rvert \otimes \lvert\boldsymbol{C}\rvert \arrow[d, "\mu_{\lvert C\rvert}"] & \lvert\boldsymbol{C}\rvert \otimes 1_C \arrow[l, "\lvert\boldsymbol{C}\rvert \otimes \eta_{\lvert C\rvert}"'] \arrow[dl, "\rho_{\lvert C\rvert}"] \\ & \lvert\boldsymbol{C}\rvert & \end{tikzcd}$
Associative Law	**Left/Right Unit Laws**

Table 4: Laws of Monoidal Structure

- The graph 2-cell $\mu_{|C|}$ satisfies an *associative law*. In the design of the IFF Foundation Ontology the correct expression of this important axiom has motivated the namespace for graphs and graph morphisms, the representation of categories as monoids in the 2-dimensional quasi-category GRAPH, and in particular the coherence axiomatization of graphs. The graph 2-cell eta satisfies two *unit laws*. Both the associative law and the two unit laws are expressed at the level of graph morphisms.

```
(15) (forall (?c (category ?c))
         (= (GPH.MOR$composition
                [(GPH.MOR$multiplication
                     [(GPH.MOR$identity (graph ?c)) (mu ?c)])
                 (mu ?c)])
            (GPH.MOR$composition
                [(GPH.MOR$composition
                     [(GPH.MOR$alpha [(graph ?c) (graph ?c) (graph ?c)])
                      (GPH.MOR$multiplication
                          [(mu ?c) (GPH.MOR$identity (graph ?c))])])
                 (mu ?c)])))

(16) (forall (?c (category ?c))
         (and (= (GPH.MOR$composition
                     [(GPH.MOR$multiplication
                          [(eta ?c) (GPH.MOR$identity (graph ?c))])
                      (mu ?c)])
                 (GPH.MOR$left (graph ?c)))
              (= (GPH.MOR$composition
                     [(GPH.MOR$multiplication
                          [(GPH.MOR$identity (graph ?c)) (eta ?c)])
                      (mu ?c)])
                 (GPH.MOR$right (graph ?c)))))
```

- Using composition and identity, the associative law and two unit laws can be expressed at the level of class functions – Table 5 represents the associative and unit laws in terms of the composition and identity functions.

Associative law:	$(m_1 \circ^{\boldsymbol{C}} m_2) \circ^{\boldsymbol{C}} m_3 = m_1 \circ^{\boldsymbol{C}} (m_2 \circ^{\boldsymbol{C}} m_3)$
Identity laws:	$id^{\boldsymbol{C}}_a \circ^{\boldsymbol{C}} m = m = m \circ^{\boldsymbol{C}} id^{\boldsymbol{C}}_b$

Table 5: Laws of Monoidal Structure Redux

Additional Categorical Structure

Particular categories may have additional structure. This is true for the categories represented in the IFF lower metalevel.

- A category $\boldsymbol{C}$ is *small* when its underlying graph is small. This is a concrete concept, since it uses concepts in the lower metalevel.

```
(17) (KIF$collection small)
     (KIF$subcollection small category)
     (forall (?c (category ?c))
         (<=> (small ?c)
              (GPH$small (graph ?c))))
```

- To each category C, there is an *opposite* category $\boldsymbol{C}^{\mathrm{op}} = \langle \boldsymbol{C}, \mu_{\boldsymbol{C}}, \eta_{\boldsymbol{C}} \rangle^{\mathrm{op}} = \langle \boldsymbol{C}^{\mathrm{op}}, \tau_{\boldsymbol{C},\boldsymbol{C}} \cdot \mu_{\boldsymbol{C}}{}^{\mathrm{op}}, \eta_{\boldsymbol{C}}{}^{\mathrm{op}} \rangle$. Since all categorical notions have their duals, the opposite category can be used to decrease the size of the axiom set. The objects of $\boldsymbol{C}^{\mathrm{op}}$ are the objects of $\boldsymbol{C}$, and the morphisms of $\boldsymbol{C}^{\mathrm{op}}$ are the morphisms of $\boldsymbol{C}$. However, the source and target of a morphism are reversed: $src(\boldsymbol{C}^{\mathrm{op}})(m) = tgt(\boldsymbol{C})(m)$ and $tgt(\boldsymbol{C}^{\mathrm{op}})(m) = src(\boldsymbol{C})(m)$. The composition is defined by $m_2 \circ^{\mathrm{op}} m_1 = m_1 \circ m_2$, and the identity is $id^{\mathrm{op}}_o = id_o$.

```
(18) (KIF$function opposite)
     (= (KIF$source opposite) category)
     (= (KIF$target opposite) category)
     (forall (?c (category ?c))
         (and (= (graph (opposite ?c)) (GPH$opposite (graph ?c))
              (= (mu (opposite ?c))
                 (GPH.MOR$composition
                     [(GPH.MOR$tau [(graph ?c) (graph ?c)])
                      (GPH.MOR$opposite (mu ?c))]))
              (= (eta (opposite ?c)) (GPH.MOR$opposite (eta ?c)))))
```

- Part of the fact that opposite forms an involution is the theorem that $(\boldsymbol{C}^{\mathrm{op}})^{\mathrm{op}} = \boldsymbol{C}$.

```
     (forall (?c (category ?c))
         (= (opposite (opposite ?c)) ?c))
```

- A category $\boldsymbol{A}$ is said to be a *subcategory* of category $\boldsymbol{B}$ when the object (morphism) class of $\boldsymbol{A}$ is a subclass of the object (morphism) class of $\boldsymbol{B}$, and the source (target) function of $\boldsymbol{A}$ is a restriction of the source (target) function of $\boldsymbol{A}$. Clearly, this provides a partial order on categories.

```
(19) (KIF$relation subcategory)
     (= (KIF$collection1 subcategory) CAT$category)
     (= (KIF$collection2 subcategory) CAT$category)
     (forall (?a (CAT$category ?a) ?b (CAT$category ?b))
         (<=> (subcategory ?a ?b)
              (GPH$subgraph (graph ?a) (graph ?b))))
```

- For any category, it is sometimes convenient to have a name for both its *object pair* and its *morphism pair* of classes.

```
(20) (KIF$function object-pair)
     (= (KIF$source object-pair) category)
     (= (KIF$target object-pair) SET.LIM.PRD2$pair)
     (forall (?c (category ?c))
         (and (= (SET.LIM.PRD2$class1 (object-pair ?c)) (object ?c))
              (= (SET.LIM.PRD2$class2 (object-pair ?c)) (object ?c))))

(21) (KIF$function object-binary-product)
     (= (KIF$source object-binary-product) category)
     (= (KIF$target object-binary-product) SET.class)
     (forall (?c (category ?c))
         (= (object-binary-product ?c)
            (SET.LIM.PRD2$binary-product (object-pair ?c))))

(22) (KIF$function morphism-pair)
     (= (KIF$source morphism-pair) category)
     (= (KIF$target morphism-pair) SET.LIM.PRD2$pair)
     (forall (?c (category ?c))
         (and (= (SET.LIM.PRD2$class1 (morphism-pair ?c)) (morphism ?c))
              (= (SET.LIM.PRD2$class2 (morphism-pair ?c)) (morphism ?c))))
```

```
(23) (KIF$function morphism-binary-product)
     (= (KIF$source morphism-binary-product) category)
     (= (KIF$target morphism-binary-product) SET.class)
     (forall (?c (category ?c))
         (= (morphism-binary-product ?c)
            (SET.LIM.PRD2$binary-product (morphism-pair ?c))))
```

- For any two objects o_1 and o_2 in a category $\boldsymbol{C}$, the *hom-class* $\boldsymbol{C}[o_1, o_2]$ consists of all morphisms with source o_1 and target o_2:

  $$\boldsymbol{C}[o_1, o_2] = \{m \mid m \in \mathit{mor}(\boldsymbol{C}),\ \mathit{src}(\boldsymbol{C})(m) = o_1 \text{ and } \mathit{tgt}(\boldsymbol{C})(m) = o_2\} \subseteq \mathit{mor}(\boldsymbol{C}).$$

```
(24) (KIF$function source-target)
     (= (KIF$source source-target) category)
     (= (KIF$target source-target) SET.FTN$function)
     (forall (?c (category ?c))
         (and (= (SET.FTN$source (source-target ?c)) (morphism ?c))
              (= (SET.FTN$target (source-target ?c)) (object-binary-product ?c))
              (= (source-target ?c)
                 ((SET.LIM.PRD2$pairing (object-pair ?c)) [(source ?c) (target
?c)]))))

(25) (KIF$function object-hom)
     (= (KIF$source object-hom) category)
     (= (KIF$target object-hom) SET.FTN$function)
     (forall (?c (category ?c))
         (and (= (SET.FTN$source (object-hom ?c)) (object-binary-product ?c))
              (= (SET.FTN$target (object-hom ?c)) (SET$power (morphism ?c)))
              (= (object-hom ?c) (SET.FTN$fiber (source-target ?c)))))

(26) (KIF$function morphism-hom)
     (= (KIF$source morphism-hom) category)
     (= (KIF$target morphism-hom) KIF$function)
     (forall (?c (category ?c))
         (and (= (KIF$source (morphism-hom ?c)) (morphism-binary-product ?c))
              (= (KIF$target (morphism-hom ?c)) SET.FTN$function)
              (forall (?m1 ((morphism c) ?m1) ?m2 ((morphism c) ?m2))
                  (and (= (SET.FTN$source ((morphism-hom ?c) [?m1 ?m2]))
                          ((object-hom ?c) [((target ?c) ?m1) ((source ?c) ?m2)]))
                       (= (SET.FTN$target ((morphism-hom ?c) [?m1 ?m2]))
                          ((object-hom ?c) [((source ?c) ?m1) ((target ?c) ?m2)]))
                       (forall (?m ((morphism c) ?m)
                           (=> (and (= ((source ?c) ?m)) ((target ?c) ?m1))
                                    (= ((source ?c) ?m)) ((source ?c) ?m2)))
                               (= (((morphism-hom ?c) [?m1 ?m2]) ?m)
                                  ((composition ?c)
                                      [((composition ?c) [?m1 ?m]) ?m2]))))))))
```

- There are classes of left-composability and right-composability, and functions of left-composition and right-composition. Left-composition by morphism m_1 is the operation: $m_2 \mapsto m_1 \cdot m_2$.

```
(27) (KIF$function left-composable)
     (= (KIF$source left-composable) category)
     (= (KIF$target left-composable) SET.FTN$function)
     (forall (?c (category ?c))
         (and (= (SET.FTN$source (left-composable ?c)) (morphism ?c))
              (= (SET.FTN$target (left-composable ?c)) (SET$power (morphism ?c)))
              (= (left-composable ?c) (REL$fiber12 (composable ?c)))))

(28) (KIF$function left-composition)
     (= (KIF$source left-composition) category)
     (= (KIF$target left-composition) KIF$function)
     (forall (?c (category ?c))
       (and (= (KIF$source (left-composition ?c)) (morphism ?c))
            (= (KIF$target (left-composition ?c)) SET.FTN$function)
            (forall (?m1 ((morphism ?c) ?m1))
                (and (= (SET.FTN$source ((left-composition ?c) ?m1))
                        ((left-composable ?c) ?m1))
                     (= (SET.FTN$target ((left-composition ?c) ?m1))
                        (morphism ?c))
                        (= ((left-composition ?c) ?m1)
                           (SET.FTN$composition
                               [((REL$fiber12-embedding (composable ?c)) ?m1)
                                (composition ?c)])))))))

(29) (KIF$function right-composable)
     (= (KIF$source right-composable) category)
     (= (KIF$target right-composable) SET.FTN$function)
     (forall (?c (category ?c))
         (and (= (SET.FTN$source (right-composable ?c)) (morphism ?c))
              (= (SET.FTN$target (right-composable ?c)) (SET$power (morphism ?c)))
              (= (right-composable ?c) (REL$fiber21 (composable ?c))))))

(30) (KIF$function right-composition)
     (= (KIF$source right-composition) category)
     (= (KIF$target right-composition) KIF$function)
     (forall (?c (category ?c))
       (and (= (KIF$source (right-composition ?c)) (morphism ?c))
            (= (KIF$target (right-composition ?c)) SET.FTN$function)
            (forall (?m2 ((morphism ?c) ?m2))
                (and (= (SET.FTN$source ((right-composition ?c) ?m2))
                        ((right-composable ?c) ?m2))
                     (= (SET.FTN$target ((right-composition ?c) ?m2))
                        (morphism ?c))
                     (= ((right-composition ?c) ?m2)
                        (SET.FTN$composition
                            [((REL$fiber21-embedding (composable ?c)) ?m2)
                             (composition ?c)])))))))
```

○ A morphism $m_1 : o_0 \rightarrow o_1$ is an *epimorphism* in a category ***C*** when it is left-cancellable – for any two parallel morphisms $m_2, m_2' : o_1 \rightarrow o_2$, the equality $\mathrm{m}_1 \circ^{\mathrm{C}} \mathrm{m}_2 = \mathrm{m}_1 \circ^{\mathrm{C}} \mathrm{m}_2'$ implies $\mathrm{m}_2 = \mathrm{m}_2'$. Equivalently, a morphism $m_1 : o_0 \rightarrow o_1$ is an epimorphism in a category ***C*** when its left composition is an injection. Dually, a morphism $m_2 : o_1 \rightarrow o_2$ is a *monomorphism* in a category ***C*** when it is right-cancellable – that is, when it is an epimorphism in C^{op}. The duality of the opposite category is used to express monomorphisms. A morphism is an *isomorphism* in a category ***C*** when it is both a monomorphism and an epimorphism.

```
(31) (KIF$function epimorphism)
     (= (KIF$source epimorphism) category)
     (= (KIF$target epimorphism) SET$class)
     (forall (?c (category ?c))
         (and (SET$subclass (epimorphism ?c) (morphism ?c))
              (forall (?m1 ((morphism ?c) ?m1))
                  (<=> ((epimorphism ?c) ?m1)
                       (SET.FTN$injection ((left-composition ?c) ?m1))))))

(32) (KIF$function monomorphism)
     (= (KIF$source monomorphism) category)
     (= (KIF$target monomorphism) SET$class)
     (forall (?c (category ?c))
         (and (SET$subclass (monomorphism ?c) (morphism ?c))
              (forall (?m2 ((morphism ?c) ?m2))
                  (<=> ((monomorphism ?c) ?m1)
                       ((epimorphism (opposite ?c)) ?m1)))))

(33) (KIF$function isomorphism)
     (= (KIF$source isomorphism) category)
     (= (KIF$target isomorphism) SET$class)
     (forall (?c (category ?c))
         (and (SET$subclass (isomorphism ?c) (morphism ?c))
              (= (isomorphism ?c)
                 (SET$binary-intersection [(epimorphism ?c) (monomorphism ?c)]))))
```

○ Two objects $o_1, o_2 \in mor(C)$ are *isomorphic* when there is an isomorphism between them; we then use the notation $o_1 \cong_C o_2$.

```
(34) (KIF$function isomorphic)
     (= (KIF$source isomorphic) category)
     (= (KIF$target isomorphic) REL.ENDO$equivalence-relation)
```

```
        (forall (?c (category ?c))
            (and (REL.ENDO$class (isomorphic ?c)) (object ?c))
                 (forall (?o1 ((object ?c) ?o1) ?o2 ((object ?c) ?o2))
                     (<=> ((isomorphic ?c) ?o1 ?o2)
                          (exists (?m ((isomorphism ?c) ?m))
                              (and (= ((source ?c) ?m) ?o1)
                                   (= ((target ?c) ?m) ?o2)))))))
```

○ A *discrete* category is a category whose morphisms are all identity morphisms – effectively, the morphism class can be identified with the object class, and the source and target functions are then identity functions. A discrete category is essentially a class (of objects).

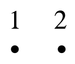
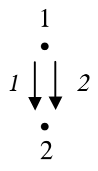
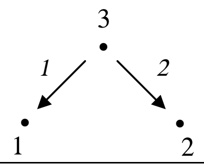
```
(35) (KIF$collection discrete)
     (KIF$subcollection discrete category)
     (forall (?c (category ?c))
         (<=> (discrete ?c)
              (and (SET.FTN$isomorphic (morphism ?c) (object ?c))
                   (= (source ?c) (target ?c))
                   (SET.FTN$bijection (source ?c)))))
```

Examples

Here are some examples of categories. We express these in an external namespace.

○ Here are some examples (Table 6) of small (in fact, finite) categories that are used as the shapes of finite colimit diagrams in a category ***C***: for a terminal ***C***-object (this uses the *empty* shape diagram), for the binary coproduct of *two* ***C***-objects, for the coequalizers of a *parallel pair* of ***C***-morphisms, and for the pushout of a *span* of ***C***-objects and ***C***-morphisms. The categories empty and two are discrete. All four free categories do not add additional morphisms (paths), other than the identity morphisms.

Table 6: Shapes for Diagrams in a Category

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

```
(36) (category empty)
     (= empty (GPH$category GPH$empty))

(37) (category two)
     (= two (GPH$category GPH$two))

(38) (category parallel-pair)
     (= parallel-pair (GPH$category GPH$parallel-pair))

(39) (category span)
     (= span (GPH$category GPH$span))
```

○ The discrete *terminal* (or *unit*) category has one object and one (identity) morphism.

```
(40) (CAT$category terminal)
     (CAT$category unit)
     (CAT$category one)
     (= unit terminal)
     (= one unit)
     (= (CAT$object terminal) SET.LIM$terminal)
     (= (CAT$morphism terminal) SET.LIM$terminal)
     (= (CAT$source terminal) (SET.FTN$identity SET.LIM$terminal))
     (= (CAT$target terminal) (SET.FTN$identity SET.LIM$terminal))
```

○ Here are examples of categories defined elsewhere, but asserted to be categories here. These are concrete concepts, since they use concepts in the lower metalevel. Two important categories are implicitly defined within the classification namespace in the IFF Model Ontology – Classification the category of classifications and infomorphisms, and Set the category of sets (small collections) and their functions. The assertion that "Classification *and* Set *are categories*" can not be made in the IFF Model Theory Ontology, a module in the lower metalevel of the IFF Foundation Ontology, since the appropriate categorical/functorial machinery is not present there. The IFF Category Theory Ontology provides that machinery.

– To make these assertions requires that we also describe or identify the components of a category. The most concise expression is in terms of the underlying graph, and the mu and eta graph 2-cells. This assumes that these have been suitably defined. Here we make these assertions in an external namespace.

```
(41) (CAT$category Set)
     (= (CAT$graph Set) set.ftn$graph)
     (= (CAT$mu Set)    set.ftn$mu)
     (= (CAT$eta Set)   set.ftn$eta)

(42) (CAT$category Classification)
     (= (CAT$graph Classification) cls.info$graph)
     (= (CAT$mu Classification)    cls.info$mu)
     (= (CAT$eta Classification)   cls.info$eta)
```

– A more detailed expression is in terms of the object and morphism sets, the source and target functions, and the composition and identity functions. Proofs of some categorical properties, such as associativity of composition, will involve getting further into the details of the specific category, in this case Classification; in particular, associativity of the composition operation, etc.

```
(CAT$category Set)
(= (CAT$object Set)       set$set)
(= (CAT$morphism Set)     set.ftn$function)
(= (CAT$source Set)       set.ftn$source)
(= (CAT$target Set)       set.ftn$target)
(= (CAT$composable Set)   set.ftn$composable)
(= (CAT$first Set)        set.ftn$first)
(= (CAT$second Set)       set.ftn$second)
(= (CAT$composition Set)  set.ftn$composition)
(= (CAT$identity Set)     set.ftn$identity)

(CAT$category Classification)
(= (CAT$object Classification)       cls$classification)
(= (CAT$morphism Classification)     cls.info$infomorphism)
(= (CAT$source Classification)       cls.info$source)
(= (CAT$target Classification)       cls.info$target)
(= (CAT$composable Classification)   cls.info$composable)
(= (CAT$first Classification)        cls.info$first)
(= (CAT$second Classification)       cls.info$second)
(= (CAT$composition Classification)  cls.info$composition)
(= (CAT$identity Classification)     cls.info$identity)
```

The Namespace of Large Functors

`FUNC`

Categories are related by functors. This namespace represents large functors. A large functor is a functor between large categories; that is, a functor whose object and morphism functions are class functions. The content of the namespace for functors is initially developed in chapter 1 section 3 of Mac Lane 1971. All terms declared and axiomatized in this namespace are listed in Table 1. The suggested prefix for this namespace is '`FUNC`', standing for functor: when used in an external namespace, all terms that originate from this namespace should be prefixed with '`FUNC`'.

Table 1: Terms introduced in the Category Theory Ontology

	Collection	Function	Other
`FUNC`	`functor`	`source target`	`composable-opspan`
	`opspan`	`graph-morphism = underlying object morphism`	`composable`
		`unique element diagonal = constant`	`instance type`
		`inclusion opposite exponent`	`instance-power`
		`category1 category2 opvertex opfirst opsecond`	
		`comma-category`	
		`objects-under-opspan objects-under`	
		`universal-morphism`	
		`composition identity`	

Basics

○ A *functor* $\boldsymbol{F} : \boldsymbol{C}_0 \rightarrow \boldsymbol{C}_1$ from *source* category $\boldsymbol{C}_0$ to *target* category $\boldsymbol{C}_1$ (Figure 1) is a morphism of categories. It consists of an *object* function between object classes of the categories, and a *morphism* function between morphism classes of the categories. A functor is a special kind of graph morphism $|\boldsymbol{F}| : |\boldsymbol{C}_0| \rightarrow |\boldsymbol{C}_1|$ – a graph morphism that preserves the monoidal properties of the categories. A functor is determined by its associated triple (source, target, underlying graph morphism).

$$\begin{tikzcd}
mor(|\boldsymbol{C}_0|) \arrow[r, shift left, "src(|\boldsymbol{C}_0|)"] \arrow[r, shift right, "tgt(|\boldsymbol{C}_0|)"'] \arrow[d, "mor(|\boldsymbol{F}|)"'] & obj(|\boldsymbol{C}_0|) \arrow[d, "obj(\boldsymbol{F})"] \\
mor(|\boldsymbol{C}_1|) \arrow[r, shift left, "src(|\boldsymbol{C}_1|)"] \arrow[r, shift right, "tgt(|\boldsymbol{C}_1|)"'] & obj(|\boldsymbol{C}_1|)
\end{tikzcd}$$

Figure 1: Functor

```
(1) (KIF$collection functor)

(2) (KIF$function source)
    (= (KIF$source source) functor)
    (= (KIF$target source) CAT$category)

(3) (KIF$function target)
    (= (KIF$source target) functor)
    (= (KIF$target target) CAT$category)

(4) (KIF$function graph-morphism)
    (KIF$function underlying)
    (= underlying graph-morphism)
    (= (KIF$source graph-morphism) functor)
    (= (KIF$target graph-morphism) GPH.MOR$graph-morphism)

(5) (forall (?f (functor ?f))
        (and (= (CAT$underlying (source ?f))
                (GPH.MOR$source (underlying ?f)))
             (= (CAT$underlying (target ?f))
                (GPH.MOR$target (underlying ?f)))))

    (forall (?f1 (functor ?f1) ?f2 (functor ?f2))
        (=> (and (= (source ?f1) (source ?f2))
                 (= (target ?f1) (target ?f2))
                 (= (graph-morphism ?f1) (graph-morphism ?f2)))
            (= ?f1 ?f2)))
```

- For convenience in the language used for functors, in axioms (5–6) we rename the *object* and *morphism* functions in the setting of functors.

```
(6) (KIF$function object)
    (= (KIF$source object) functor)
    (= (KIF$target object) SET.FTN$function)
    (forall (?f (functor ?f))
        (= (object ?f) (GPH.MOR$object (graph-morphism ?f))))

(7) (KIF$function morphism)
    (= (KIF$source morphism) function)
    (= (KIF$target morphism) SET.FTN$function)
    (forall (?f (functor ?f))
        (= (morphism ?f) (GPH.MOR$morphism (graph-morphism ?f))))
```

Table 2: Preservation of monoidal structure

$$\begin{tikzcd} \lvert \boldsymbol{C}_0 \rvert \otimes \lvert \boldsymbol{C}_0 \rvert \arrow[r, "\mu_{\lvert C0 \rvert}"] \arrow[d, "\lvert \boldsymbol{F} \rvert \otimes \lvert \boldsymbol{F} \rvert"'] & \lvert \boldsymbol{C}_0 \rvert \arrow[d, "\lvert \boldsymbol{F} \rvert"] \\ \lvert \boldsymbol{C}_1 \rvert \otimes \lvert \boldsymbol{C}_1 \rvert \arrow[r, "\mu_{\lvert C1 \rvert}"'] & \lvert \boldsymbol{C}_1 \rvert \end{tikzcd}$$	$$\begin{tikzcd} 1_{obj(\lvert C0 \rvert)} \arrow[r, "\eta_{\lvert C0 \rvert}"] \arrow[d, "1_{obj(\lvert F \rvert)}"'] & \lvert \boldsymbol{C}_0 \rvert \arrow[d, "\lvert \boldsymbol{F} \rvert"] \\ 1_{obj(\lvert C1 \rvert)} \arrow[r, "\eta_{\lvert C1 \rvert}"'] & \lvert \boldsymbol{C}_1 \rvert \end{tikzcd}$$
Preservation of composition	**Preservation of identity**

- A functor must preserve monoidal properties – it must preserve identities and compositions in the sense of the commutative diagrams in Table 2. These are commutative diagrams of graph morphisms. The commutative diagram on the left represents preservation of composition, and the commutative diagram on the right represents preservation of identity.

```
(8) (forall (?f (functor ?f))
        (= (GPH.MOR$composition [(CAT$mu (source ?f)) (graph-morphism ?f)])
           (GPH.MOR$composition
               [(GPH.MOR$multiplication [(graph-morphism ?f) (graph-morphism ?f)])
                (CAT$mu (target ?f))])))

(9) (forall (?f (functor ?f))
        (= (GPH.MOR$composition [(CAT$eta (source ?f)) (graph-morphism ?f)])
           (GPH.MOR$composition [(GPH.MOR$unit (object ?f)) (CAT$eta (target ?f))])))
```

- Using composition and identity, the associative and unit laws could also be expressed at the level of class functions as in Table 2, which is derivative – it represents the preservation of monoidal structure in terms of the composition and identity functions.

Table 2: Laws of Monoidal Structure Redux

Associative law:	$mor(\lvert \boldsymbol{F} \rvert)(m_1 \circ^0 m_2) = mor(\lvert \boldsymbol{F} \rvert)(m_1) \circ^1 mor(\lvert \boldsymbol{F} \rvert)(m_2)$ for all composable pairs of morphisms $m_1, m_2 \in mor(\boldsymbol{C}_0)$
Identity laws:	$mor(\lvert \boldsymbol{F} \rvert)(id^0_o) = id^1_{obj(\lvert \boldsymbol{F} \rvert)(o)}$ for all objects $o \in obj(\boldsymbol{C}_0)$

```
(forall (?f (functor ?f))
    (= (SET.FTN$composition [(CAT$composition (source ?f)) (morphism ?f)])
       (SET.FTN$composition
           [((SET.LIM.PBK$pairing (CAT$composable-opspan (target ?f)))
                [(SET.FTN$composition [(CAT$first (source ?f)) (morphism ?f)])
                 (SET.FTN$composition [(CAT$second (source ?f)) (morphism ?f)])]])
           (CAT$composition (target ?f))]))))

(forall (?f (functor ?f))
    (= (SET.FTN$composition [(CAT$identity (source ?f)) (morphism ?f)])
       (SET.FTN$composition [(object ?f) (CAT$identity (target ?f))])))
```

Additional Functorial Structure

○ Given any category $\boldsymbol{C}$, there is a *unique* functor $!_{\boldsymbol{C}} : \boldsymbol{C} \rightarrow 1$ to the terminal category – the object and morphism functions are the unique class functions to the terminal class.

```
(10) (KIF$function unique)
     (= (KIF$source unique) CAT$category)
     (= (KIF$target unique) functor)
     (forall (?c (CAT$category ?c))
         (and (= (source (unique ?c)) ?c)
              (= (target (unique ?c)) CAT$terminal)
              (= (object (unique ?c)) (SET.LIM$unique (CAT$object ?c)))
              (= (morphism (unique ?c)) (SET.LIM$unique (CAT$morphism ?c)))))
```

○ For each category $\boldsymbol{C}$ and each object $o \in \boldsymbol{C}$ there is an *element* functor $\mathsf{elmt}_{\boldsymbol{C}}(o) : 1 \rightarrow \boldsymbol{C}$.

```
(11) (KIF$function element)
     (= (KIF$source element) CAT$category)
     (= (KIF$target element) KIF$function)
     (forall (?c (CAT$category ?c))
         (and (= (KIF$source (element ?c)) (CAT$object ?c))
              (= (KIF$target (element ?c)) functor)
              (forall (?o ((CAT$object ?c) ?o))
                  (and (= (source ((element ?c) ?o)) CAT$terminal)
                       (= (target ((element ?c) ?o)) ?c)
                       (= ((object ((element ?c) ?o)) 1) ?o)
                       (= ((morphism ((element ?c) ?o)) 1)
                          ((CAT$identity ?c) ?o))))))
```

○ Given any category $\boldsymbol{C}$ (to be used as a base category in the colimit namespace) and any category $\boldsymbol{J}$ (to be used as a shape category in the colimit namespace), the *diagonal* (or *constant*) *functor* $\Delta_{\boldsymbol{J},\boldsymbol{C}} : \boldsymbol{C} \rightarrow \boldsymbol{C}^{\boldsymbol{J}}$ maps an object $o \in \mathsf{obj}(\boldsymbol{C})$ to an associated constant functor $\Delta_{\boldsymbol{J},\boldsymbol{C}}(o) : \boldsymbol{J} \rightarrow \boldsymbol{C}$, which is defined as the functor composition $\Delta_{\boldsymbol{J},\boldsymbol{C}}(o) = !_{\boldsymbol{J}} \circ \mathsf{elmt}_{\boldsymbol{C}}(o)$ – it maps each object $j \in \mathsf{obj}(\boldsymbol{J})$ to the object $o \in \mathsf{obj}(\boldsymbol{C})$ and maps each morphism $n \in \mathsf{mor}(\boldsymbol{J})$ to the identity morphism at o.

```
(12) (KIF$function diagonal)
     (KIF$function constant)
     (= constant diagonal)
     (= (KIF$source diagonal)
        (KIF$binary-product [CAT$category CAT$category]))
     (= (KIF$target diagonal) KIF$function)
     (forall (?j (CAT$category ?j) ?c (CAT$category ?c))
         (and (= (KIF$source (diagonal [?j ?c])) (CAT$object ?c))
              (= (KIF$target (diagonal [?j ?c])) functor)
              (forall (?o ((CAT$object ?c) ?o))
                  (and (= (source ((diagonal [?j ?c]) ?o)) ?j)
                       (= (target ((diagonal [?j ?c]) ?o)) ?c)
                       (= ((diagonal [?j ?c]) ?o)
                          (composition [(unique ?j) ((element ?c) ?o)]))))))
```

○ If category $\boldsymbol{A}$ is a subcategory of category $\boldsymbol{B}$, there is an *inclusion* functor $\mathsf{incl}_{\boldsymbol{A},\boldsymbol{B}} : \boldsymbol{A} \rightarrow \boldsymbol{B}$ whose object and morphism functions are the subclass inclusion functions.

```
(13) (KIF$relation inclusion)
     (= (KIF$source subcategory) (KIF$extent CAT$subcategory))
     (= (KIF$target subcategory) functor)
     (forall (?a (CAT$category ?a) ?b (CAT$category ?b) (subcategory ?a ?b))
         (and (= (source (inclusion [?a ?b])) ?a)
              (= (target (inclusion [?a ?b])) ?b)
              (= (graph-morphism (inclusion [?a ?b]))
                 (GPH.MOR$inclusion [(graph ?a) (graph ?b)]))))
```

- To each functor $\boldsymbol{F} : \boldsymbol{C} \rightarrow \boldsymbol{C}'$, there is an *opposite functor* $\boldsymbol{F}^{\mathrm{op}} : \boldsymbol{C}^{\mathrm{op}} \Rightarrow \boldsymbol{C}'^{\mathrm{op}}$. The underlying graph morphism of $\boldsymbol{F}^{\mathrm{op}}$ is the opposite $|\boldsymbol{F}^{\mathrm{op}}| = |\boldsymbol{F}|^{\mathrm{op}} : |\boldsymbol{C}_0|^{\mathrm{op}} \rightarrow |\boldsymbol{C}_1|^{\mathrm{op}}$: the object function of $\boldsymbol{F}^{\mathrm{op}}$ is the object function of $\boldsymbol{F}$, and the morphism function of $\boldsymbol{F}^{\mathrm{op}}$ is the morphism function of $\boldsymbol{F}$. However, the source and target categories are the opposite: $src(\boldsymbol{F}^{\mathrm{op}}) = src(\boldsymbol{F})^{\mathrm{op}}$ and $tgt(\boldsymbol{F}^{\mathrm{op}}) = tgt(\boldsymbol{F})^{\mathrm{op}}$.

```
(14) (KIF$function opposite)
     (= (KIF$source opposite) functor)
     (= (KIF$target opposite) functor)
     (forall (?f (functor ?f))
         (and (= (source (opposite ?f)) (CAT$opposite (source ?f)))
              (= (target (opposite ?f)) (CAT$opposite (target ?f)))
              (= (graph-morphism (opposite ?f))
                 (GPH.MOR$opposite (graph-morphism ?h)))))
```

An immediate theorem is that the opposite of the opposite of a functor is the original functor.

```
     (forall (?f (functor ?f))
         (= (opposite (opposite ?f)) ?f))
```

- For any pair of categories $\boldsymbol{C}_1$ and $\boldsymbol{C}_2$ in there is an *exponent* collection of functors whose source category is $\boldsymbol{C}_1$ and whose target category is $\boldsymbol{C}_2$.

```
(15) (KIF$function exponent)
     (= (KIF$source exponent) (KIF$binary-product [CAT$category CAT$category]))
     (= (KIF$target exponent) KIF$collection)
     (forall (?c1 (CAT$category ?c1) ?c2 (CAT$category ?c2))
         (and (KIF$subcollection (exponent [?c1 ?c2]) functor)
              (forall (?f (functor ?f))
                  (<=> ((exponent [?c1 ?c2]) ?f)
                       (and (= (source ?f) ?c1)
                            (= (target ?f) ?c2))))))
```

Comma Categories

Comma categories are developed in section 2.6 in Mac Lane 1971.

- An *opspan* of functors consists of two functors $\boldsymbol{F} : \boldsymbol{A} \rightarrow \boldsymbol{C}$ and $\boldsymbol{G} : \boldsymbol{B} \rightarrow \boldsymbol{C}$ with a common target category $\boldsymbol{C}$. An opspan is determined by its opfirst and opsecond pair of functors.

```
(16) (KIF$collection opspan)

(17) (KIF$function category1)
     (= (KIF$source category1) opspan)
     (= (KIF$target category1) CAT$category)

(18) (KIF$function category2)
     (= (KIF$source category2) opspan)
     (= (KIF$target category2) CAT$category)

(19) (KIF$function opvertex)
     (= (KIF$source opvertex) opspan)
     (= (KIF$target opvertex) CAT$category)

(20) (KIF$function opfirst)
     (= (KIF$source opfirst) opspan)
     (= (KIF$target opfirst) functor)

(21) (KIF$function opsecond)
     (= (KIF$source opsecond) opspan)
     (= (KIF$target opsecond) functor)

(22) (forall (?s (opspan ?s))
         (and (= (SET.FTN$source (opfirst ?s)) (category1 ?s))
              (= (SET.FTN$source (opsecond ?s)) (category2 ?s))
              (= (SET.FTN$target (opfirst ?s)) (opvertex ?s))
              (= (SET.FTN$target (opsecond ?s)) (opvertex ?s))))
     (forall (?s (opspan ?s) ?t (opspan ?t))
         (=> (and (= (opfirst ?s) (opfirst ?t))
                  (= (opsecond ?s) (opsecond ?t)))
             (= ?s ?t)))
```

○ Each opspan of functors determines a *comma category* $(\boldsymbol{F} \downarrow \boldsymbol{G})$, whose objects are triples $\langle a, m, b\rangle$ with $a \in \mathit{obj}(\boldsymbol{A})$, $b \in \mathit{obj}(\boldsymbol{B})$ and $m : \boldsymbol{F}(a) \rightarrow \boldsymbol{G}(b)$, and whose morphisms

$$\langle u, v\rangle : \langle a_1, m_1, b_1\rangle \rightarrow \langle a_2, m_2, b_2\rangle$$

are pairs of morphisms $u : a_1 \rightarrow a_2$ and $v : b_1 \rightarrow b_2$ from the source categories that satisfy the commutative Diagram 1.

$$\begin{tikzcd}
\boldsymbol{F}(a_1) \arrow[r, "m_1"] \arrow[d, "\boldsymbol{F}(u)"'] & \boldsymbol{G}(b_1) \arrow[d, "\boldsymbol{G}(v)"] \\
\boldsymbol{F}(a_2) \arrow[r, "m_2"'] & \boldsymbol{G}(b_2)
\end{tikzcd}$$

Diagram 1: Comma Category

```
(23) (KIF$function comma-category)
     (= (KIF$source comma-category) opspan)
     (= (KIF$target comma-category) CAT$category)
     (forall (?s (opspan ?s))
         (and (forall (?o)
                  (<=> ((CAT$object (comma-category ?s)) ?o)
                       (and (KIF$triple ?o)
                            ((CAT$object (category1 ?s)) (?o 1))
                            ((CAT$morphism (opvertex ?s)) (?o 2))
                            ((CAT$object (category2 ?s)) (?o 3))
                            (= ((CAT$source (opvertex ?s)) (?o 2))
                               ((object (opfirst ?s)) (?o 1)))
                            (= ((CAT$target (opvertex ?s)) (?o 2))
                               ((object (opsecond ?s)) (?o 3))))))
              (forall (?m)
                  (<=> ((CAT$morphism (comma-category ?s)) ?m))
                       (and (KIF$pair ?m)
                            ((CAT$morphism (category1 ?s)) (?m 1))
                            (= ((CAT$source (category1 ?s)) (?m 1))
                               ((CAT$source ?m) 1))
                            (= ((CAT$target (category1 ?s)) (?m 1))
                               ((CAT$target ?m) 1))
                            ((CAT$morphism (category2 ?s)) (?m 2))
                            (= ((CAT$source (category2 ?s)) (?m 2))
                               ((CAT$source ?m) 3))
                            (= ((CAT$target (category2 ?s)) (?m 2))
                               ((CAT$target ?m) 3))
                            (= (((CAT$composition (opvertex ?s))
                                   [((CAT$source ?m) 2) (?m 2)])
                               (((CAT$composition (opvertex ?s))
                                   [(?m 1) ((CAT$target ?m) 2)])))))))

(24) (KIF$function objects-under-opspan)
     (= (KIF$source objects-under-opspan) functor)
     (= (KIF$target objects-under-opspan) KIF$function)
     (forall (?g (functor ?g))
         (and (= (KIF$source (objects-under-opspan ?g)) (CAT$object (target ?g)))
              (= (KIF$target (objects-under-opspan ?g) opspan)
              (forall (?c ((CAT$object (target ?g)) ?c))
                  (and (= (category1 ((objects-under-opspan ?g) ?c)) CAT$terminal)
                       (= (category2 ((objects-under-opspan ?g) ?c)) (source ?g))
                       (= (opvertex ((objects-under-opspan ?g) ?c)) (target ?g))
                       (= (opfirst ((objects-under-opspan ?g) ?c))
                          ((element (target ?g)) ?c))
                       (= (opsecond ((objects-under-opspan ?g) ?c)) ?g)))))

(25) (KIF$function objects-under)
     (= (KIF$source objects-under) functor)
     (= (KIF$target objects-under) KIF$function)
     (forall (?g (functor ?g))
         (and (= (KIF$source (objects-under ?g)) (CAT$object (target ?g)))
              (= (KIF$target (objects-under ?g)) CAT.category)
              (forall (?c ((CAT$object (target ?g)) ?c))
                  (= ((objects-under ?g) ?c)
                     (comma-category ((objects-under-opspan ?g) ?c))))))
```

○ For any functor $\boldsymbol{U} : \boldsymbol{B} \rightarrow \boldsymbol{A}$ and any object $a \in \mathsf{obj}(\boldsymbol{A})$, a *universal morphism* from a to $\boldsymbol{U}$ (Diagram 2) is a pair $\langle m_a, \tilde{a} \rangle$ consisting of an object $\tilde{a} \in \mathsf{obj}(\boldsymbol{B})$ and a morphism $m_a : a \rightarrow \boldsymbol{U}(\tilde{a})$, such that for every pair $\langle m, b \rangle$ consisting of an object $b \in \mathsf{obj}(\boldsymbol{B})$ and a morphism $m : a \rightarrow \boldsymbol{U}(b)$, there is a unique morphism $m' : \tilde{a} \rightarrow b$ of $\boldsymbol{B}$ with $m_a \cdot_{\boldsymbol{A}} \boldsymbol{U}(m') = m$. Equivalently, a universal morphism is an initial object in the comma category $(a \downarrow \boldsymbol{U})$.

$$\begin{tikzcd} a \arrow[r, "m_a"] \arrow[dr, "m"'] & \boldsymbol{U}(\tilde{a}) \arrow[d, "\boldsymbol{U}(m')"] \\ & \boldsymbol{U}(b) \end{tikzcd}$$

Diagram 2: Universal Morphism

For an arbitrary pair $\langle U, a \rangle$ the universal morphism may not exist – in the code below the term '`COL$initial`' refers to a possibly empty class of objects.

```
(26) (KIF$function universal-morphism)
     (= (KIF$source universal-morphism) functor)
     (= (KIF$target universal-morphism) KIF$function)
     (forall (?u (functor ?u))
         (and (= (KIF$source (universal-morphism ?u)) (CAT$object (target ?u)))
              (= (KIF$target (universal-morphism ?u)) SET$class)
              (forall (?a ((CAT$object (target ?u)) ?a))
                  (= ((universal-morphism ?u) ?a)
                     (COL$initial ((objects-under ?u) ?a))))))
```

Quasi-Category Structure

○ A pair of functors $\boldsymbol{F}$ and $\boldsymbol{G}$ is *composable* when the target category of $\boldsymbol{F}$ is the source category of $\boldsymbol{G}$. For any composable pair of functors $\boldsymbol{F} : \boldsymbol{C}_0 \rightarrow \boldsymbol{C}_1$ and $\boldsymbol{G} : \boldsymbol{C}_1 \rightarrow \boldsymbol{C}_2$ there is a *composition functor* $\boldsymbol{F} \circ \boldsymbol{G} : \boldsymbol{C}_0 \rightarrow \boldsymbol{C}_2$. It is defined in terms of the underlying graph morphisms – object and morphism functions are the composition functions of the object and morphism functions of the component functors, respectively.

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

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

(29) (KIF$function composition)
     (= (KIF$source composition) (KIF$pullback composable-opspan))
     (= (KIF$target composition) functor)
     (forall (?f1 (functor ?f1) ?f2 (functor ?f2) (composable ?f1 ?f2))
         (and (= (source (composition [?f1 ?f2])) (source ?f1))
              (= (target (composition [?f1 ?f2])) (target ?f2))
              (= (graph-morphism (composition ?f1 ?f2))
                 (GPH.MOR$composition [(graph-morphism ?f1) (graph-morphism ?f2)]))))
```

○ For any category $\boldsymbol{C}$ there is an *identity functor* $\mathsf{id}_{\boldsymbol{C}} : \boldsymbol{C} \rightarrow \boldsymbol{C}$ on that category. Its underlying graph morphisms is the identity on the underlying graph – object and morphism functions are the identity functions on the object and morphism sets of that category, respectively.

```
(30) (KIF$function identity)
     (= (KIF$source identity) CAT$category)
     (= (KIF$target identity) functor)
     (forall (?c (CAT$category ?c))
         (and (= (source (identity ?c)) ?c)
              (= (target (identity ?c)) ?c)
              (= (graph-morphism (identity ?c))
                 (GPH.MOR$identity (graph ?c)))))
```

○ It can be shown that functor composition satisfies the following associative law

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

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

$id_{C0} \circ \boldsymbol{F} = \boldsymbol{F}$ and $\boldsymbol{F} = \boldsymbol{F} \circ id_{C1}$

for any functor $\boldsymbol{F}: \boldsymbol{C}_0 \rightarrow \boldsymbol{C}_1$ with source category $\boldsymbol{C}_0$ and target category $\boldsymbol{C}_1$. Categories as objects and functors as morphisms form a quasi-category ("quasi" since this is at the level of collections in foundations). This has the following expression in an external namespace.

```
(forall (?f1 (FUNC$functor ?f1) ?f2 (FUNC$functor ?f2) ?f3 (FUNC$functor ?f3)
         (composable ?f1 ?f2) (composable ?f2 ?f3))
    (= (FUNC$composition [(FUNC$composition [?f1 ?f2]) ?f3])
       (FUNC$composition [?f1 (FUNC$composition [?f2 ?f3])])))

(forall (?f (FUNC$functor ?f))
    (and (= (FUNC$composition [(FUNC$identity (FUNC$source ?f)) ?f]) ?f)
         (= (FUNC$composition [?f (FUNC$identity (FUNC$target ?f))]) ?f)))
```

Examples

Here are examples of functors defined elsewhere, but asserted to be functors here.

○ Three important functors are implicitly defined within the classification namespace in the IFF Model Theory Ontology. These are concrete concepts, since they use concepts in the lower metalevel.

- The class functions '`cls$instance`' and '`cls.info$instance`' represent the object and morphism components of the *instance functor* *inst* : Classification → Set^{op} (the opposite of the category Set) – the object function takes a classification to its instance set and the morphism function takes an infomorphism to its instance function.
- Dually, the SET functions '`cls$type`' and '`cls.info$type`' represent the object and morphism components of the *type functor* *typ* : Classification → Set – the object function takes a classification to its type set and the morphism function takes an infomorphism to its type function.
- The class functions '`cls$instance-power`' and '`cls.info$instance-power`' represent the object and morphism components of the *instance power functor* *pow* : Set^{op} → Classification.

The assertion that "*inst*, *typ* and *pow* are functors" could not be made in the IFF Model Theory Ontology located in the lower metalevel of the IFF Foundation Ontology, since the appropriate functorial machinery is not present there. The Category Theory Ontology provides that machinery. To make that assertion requires that we also describe or identify the components of a functor: the source and target categories, and the underlying graph morphism (object and morphism functions). Here we make these assertions in an external namespace. Proofs of some functorial properties, such as preservation of associativity, will involve getting further into the details of the specific category, in this case Classification; in particular, the instance and type function components of an infomorphism, and the associativity of '`set.ftn$composition`'.

```
(30) (FUNC$functor instance)
     (= (FUNC$source instance)       Classification)
     (= (FUNC$target instance)       (CAT$opposite Set))
     (= (FUNC$underlying instance)   cls.info$instance-graph-morphism)
         (= (FUNC$object instance)   cls$instance)
         (= (FUNC$morphism instance) cls.info$instance)

(31) (FUNC$functor type)
     (= (FUNC$source type)       Classification)
     (= (FUNC$target type)       Set)
     (= (FUNC$underlying type)   cls.info$type-graph-morphism)
         (= (FUNC$object type)   cls$type)
         (= (FUNC$morphism type) cls.info$type)

(32) (FUNC$functor instance-power)
     (= (FUNC$source instance-power)      (CAT$opposite Set))
     (= (FUNC$target instance-power)       Classification)
     (= (FUNC$underlying instance-power)   cls.info$instance-power-graph-morphism)
         (= (FUNC$object instance-power)   cls$instance-power)
         (= (FUNC$morphism instance-power) cls.info$instance-power)
```

The Namespace of Large Natural Transformations

`NAT`

A natural transformation is a morphism of functors. This namespace represents large natural transformations. A natural transformation is large when its source and target functors are large. The content of the namespace for natural transformations is initially developed in chapter 1 section 4 of Mac Lane 1971. All terms declared and axiomatized in this namespace are listed in Table 1. The suggested prefix for this namespace is '`NAT`', standing for natural transformation: when used in an external namespace, all terms that originate from this namespace should be prefixed with '`NAT`'.

Table 1: Terms introduced in the Category Theory Ontology

	Collection	Function	Other

`NAT`	`natural-transformation`	`source-functor target-functor`
		`source-category target-category`
		`component`
		`vertical-composition`
		`vertical-identity`
		`horizontal-composition`
		`horizontal-identity`
		`diagonal`
			`vertically-composable-opspan`
			`vertically-composable`
			`horizontally-composable-opspan`
			`horizontally-composable`

Natural Transformations

○ Suppose that two functors $\boldsymbol{F}_0, \boldsymbol{F}_1 : \boldsymbol{C}_0 \rightarrow \boldsymbol{C}_1$ share a common source category $\boldsymbol{C}_0$ and a common target category $\boldsymbol{C}_1$. A natural transformation τ from source functor $\boldsymbol{F}_0$ to target functor $\boldsymbol{F}_1$, written 1-dimensionally as $\tau : \boldsymbol{F}_0 \Rightarrow \boldsymbol{F}_1 : \boldsymbol{C}_0 \rightarrow \boldsymbol{C}_1$ or visualized 2-dimensionally in Figure 1, is a collection of morphisms in the target category parameterized by objects in the source category that link the functorial images:

$$\begin{tikzcd} \boldsymbol{C}_0 \arrow[r, bend left, "\boldsymbol{F}_0"] \arrow[r, bend right, "\boldsymbol{F}_1"'] \arrow[r, phantom, "\Downarrow \tau"] & \boldsymbol{C}_1 \end{tikzcd}$$

Figure 1: Natural Transformation

$$\tau = \{\tau_o : \boldsymbol{F}_0(o) \rightarrow \boldsymbol{F}_1(o) \mid o \in \mathit{obj}(\boldsymbol{C}_0)\}.$$

A natural transformation is determined by its (source-functor, target-functor, component) triple.

```
(1) (KIF$collection natural-transformation)

(2) (KIF$function source-functor)
    (= (KIF$source source-functor) natural-transformation)
    (= (KIF$target source-functor) FUNC$functor)

(3) (KIF$function target-functor)
    (= (KIF$source target-functor) natural-transformation)
    (= (KIF$target target-functor) FUNC$functor)

(4) (KIF$function source-category)
    (= (KIF$source source-category) natural-transformation)
    (= (KIF$target source-category) CAT$category)
    (forall (?tau (natural-transformation ?tau))
        (and (= (FUNC$source (source-functor ?tau))
                (source-category ?tau))
             (= (FUNC$target (source-functor ?tau))
                (target-category ?tau))))

(5) (KIF$function target-category)
    (= (KIF$source target-category) natural-transformation)
    (= (KIF$target target-category) CAT$category)
    (forall (?tau (natural-transformation ?tau))
        (and (= (FUNC$source (target-functor ?tau))
                (source-category ?tau))
             (= (FUNC$target (target-functor ?tau))
                (target-category ?tau))))

(6) (KIF$function component)
```

```
        (= (KIF$source component) natural-transformation)
        (= (KIF$target component) SET.FTN$function)
        (forall (?tau (natural-transformation ?tau))
            (and (= (SET.FTN$source (component ?tau))
                    (CAT$object (source-category ?tau)))
                 (= (SET.FTN$target (component ?tau))
                    (CAT$morphism (target-category ?tau)))))

        (forall (?tau1 (natural-transformation ?tau1)
                 ?tau2 (natural-transformation ?tau2))
            (=> (and (= (source-functor ?tau1) (source-functor ?tau2))
                     (= (target-functor ?tau1) (target-functor ?tau2))
                     (= (component ?tau1) (component ?tau2)))
                (=?tau1 ?tau2)))
```

○ The source and target objects of the components of a natural transformation are image objects of the source and target functors:

$$\mathsf{src}(\boldsymbol{C}_1)(\tau(o)) = \mathsf{obj}(\boldsymbol{F}_0)(o) \text{ and } \mathsf{tgt}(\boldsymbol{C}_1)(\tau(o)) = \mathsf{obj}(\boldsymbol{F}_1)(o)$$

for any object $o \in \mathsf{obj}(\boldsymbol{C}_0)$.

```
    (7) (forall (?tau (natural-transformation ?tau))
            (and (= (SET.FTN$composition
                        [(component ?tau) (CAT$source (target-category ?tau))])
                    (FUNC$object (source-functor ?tau)))
                 (= (SET.FTN$composition
                        [(component ?tau) (CAT$target (target-category ?tau))])
                    (FUNC$object (target-functor ?tau)))))
```

○ The components of a natural transformation interact with the functorial images by satisfying the *fundamental property* (commutative Diagram 1):

$$\mathsf{mor}(\boldsymbol{F}_0)(m) \circ^1 \tau(\partial_1{}^0(m)) = \tau(\partial_0{}^0(m)) \circ^1 \mathsf{mor}(\boldsymbol{F}_1)\,(m)$$

for any morphism $m \in \mathsf{mor}(\boldsymbol{C}_0)$. Here is the (somewhat complicated) KIF representation for this interaction, which uses pullback pairing with respect to the composition opspan of the target category of the natural transformation. This is the logical KIF formalization of the fundamental property of natural transformations, expressing the commutative Diagram 1.

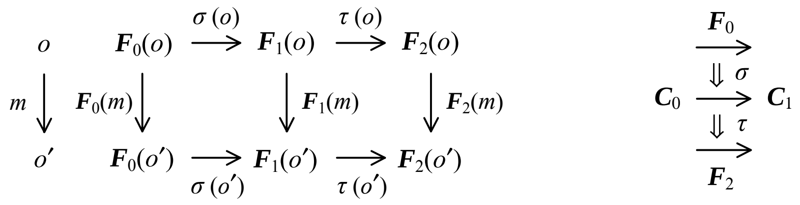
$$\begin{tikzcd}
o \arrow[d, "m"'] & F_0(o) \arrow[r, "\tau(o)"] \arrow[d, "F_0(m)"'] & F_1(o) \arrow[d, "F_1(m)"] \\
o' & F_0(o') \arrow[r, "\tau(o')"'] & F_1(o')
\end{tikzcd}$$

Diagram 1: Natural Transformation

```
    (8) (forall (?tau (natural-transformation ?tau))
            (= (SET.FTN$composition
                   [((SET.LIM.PBK$pairing (CAT$composable-opspan (target-category ?tau)))
                        [(SET.FTN$composition
                             [(CAT$source (source-category ?tau)) (component ?tau)])
                         (FUNC$morphism (target-functor ?tau))])
                    (CAT$composition (target-category ?tau))])
               (SET.FTN$composition
                   [((SET.LIM.PBK$pairing (CAT$composable-opspan (target-category ?tau)))
                        [(FUNC$morphism (source-functor ?tau))
                         (SET.FTN$composition
                             [(CAT$target (source-category ?tau)) (component ?tau)])])
                    (CAT$composition (target-category ?tau))])))
```

2-Dimensional Category Structure

- A pair of natural transformations σ and τ is *vertically composable* when the target functor of the first is the source functor of the second, $\sigma : \boldsymbol{F}_0 \Rightarrow \boldsymbol{F}_1 : \boldsymbol{C}_0 \rightarrow \boldsymbol{C}_1$ and $\tau : \boldsymbol{F}_1 \Rightarrow \boldsymbol{F}_2 : \boldsymbol{C}_0 \rightarrow \boldsymbol{C}_1$. The *vertical composition* $\sigma \bullet \tau : \boldsymbol{F}_0 \Rightarrow \boldsymbol{F}_2 : \boldsymbol{C}_0 \rightarrow \boldsymbol{C}_1$ of two vertically composable natural transformations (Diagram 2) is defined as morphism composition in the target category: $\sigma \bullet \tau\,(o) = \sigma(o) \cdot \tau(o)$ for any object $o \in obj(\boldsymbol{C}_0)$.



$$\begin{tikzcd}
o \arrow[d, "m"'] & \boldsymbol{F}_0(o) \arrow[r, "\sigma(o)"] \arrow[d, "\boldsymbol{F}_0(m)"'] & \boldsymbol{F}_1(o) \arrow[r, "\tau(o)"] \arrow[d, "\boldsymbol{F}_1(m)"] & \boldsymbol{F}_2(o) \arrow[d, "\boldsymbol{F}_2(m)"] \\
o' & \boldsymbol{F}_0(o') \arrow[r, "\sigma(o')"'] & \boldsymbol{F}_1(o') \arrow[r, "\tau(o')"'] & \boldsymbol{F}_2(o')
\end{tikzcd}
\qquad
\begin{tikzcd}
\boldsymbol{C}_0 \arrow[r, bend left=50, "\boldsymbol{F}_0"] \arrow[r, "\boldsymbol{F}_1" description] \arrow[r, bend right=50, "\boldsymbol{F}_2"'] & \boldsymbol{C}_1
\end{tikzcd}$$

$\Downarrow \sigma$ $\Downarrow \tau$

Diagram 2: Vertical composition

```
(9) (KIF$opspan vertically-composable-opspan)
    (= composable-opspan [target-functor source-functor])

(10) (KIF$relation vertically-composable)
     (= (KIF$collection1 vertically-composable) natural-transformation)
     (= (KIF$collection2 vertically-composable) natural-transformation)
     (= (KIF$extent vertically-composable)
        (KIF$pullback vertically-composable-opspan))

(11) (KIF$function vertical-composition)
     (= (KIF$source vertical-composition)
        (KIF$pullback vertically-composable-opspan))
     (= (KIF$target vertical-composition) natural-transformation)
     (forall (?sigma (natural-transformation ?sigma)
              ?tau (natural-transformation ?tau) (composable ?sigma ?tau))
         (and (= (source-functor (vertical-composition [?sigma ?tau]))
                 (source-functor ?sigma))
              (= (target-functor (vertical-composition [?sigma ?tau]))
                 (target-functor ?tau))
              (= (source-category (vertical-composition [?sigma ?tau]))
                 (source-category ?sigma))
              (= (target-category (vertical-composition [?sigma ?tau]))
                 (target-category ?sigma))
              (= (component (vertical-composition [?sigma ?tau]))
                 (SET.FTN$composition
                     [((SET.LIM.PBK$pairing
                           (CAT$composable-opspan (target-category ?sigma)))
                           [(component ?sigma) (component ?tau)])
                      (CAT$composition (target-category ?sigma))]))))
```

- For any functor $\boldsymbol{F} : \boldsymbol{C}_0 \rightarrow \boldsymbol{C}_1$ there is a *vertical identity* natural transformation $1_{\boldsymbol{F}} : \boldsymbol{F} \Rightarrow \boldsymbol{F} : \boldsymbol{C}_0 \rightarrow \boldsymbol{C}_1$ defined in terms of identity morphisms in the target category: $1_{\boldsymbol{F}}\,(o) = id_{obj(\boldsymbol{F})(o)}$ for any object $o \in obj(\boldsymbol{C}_0)$.

```
(12) (KIF$function vertical-identity)
     (= (KIF$source vertical-identity) FUNC$functor)
     (= (KIF$target vertical-identity) natural-transformation)
     (forall (?f (FUNC$functor ?f))
         (and (= (source-functor (vertical-identity ?f)) ?f)
              (= (target-functor (vertical-identity ?f)) ?f)
              (= (component (vertical-identity ?f))
                 (SET.FTN$composition
                     [(FUNC$object ?f) (CAT$identity (FUNC$target ?f))]))))
```

- A pair of natural transformations σ and τ is *horizontally composable* when the target category of the first is the source category of the second, $\sigma : \boldsymbol{F}_0 \Rightarrow \boldsymbol{F}_1 : \boldsymbol{C}_0 \rightarrow \boldsymbol{C}_1$ and $\tau : \boldsymbol{G}_0 \Rightarrow \boldsymbol{G}_1 : \boldsymbol{C}_1 \rightarrow \boldsymbol{C}_2$. The *hori-*

zontal composition $\sigma \circ \tau : \boldsymbol{F}_0 \circ \boldsymbol{G}_0 \Rightarrow \boldsymbol{F}_1 \circ \boldsymbol{G}_1 : \boldsymbol{C}_0 \rightarrow \boldsymbol{C}_2$ of two horizontally composable natural transformations (Diagram 3) is defined by pasting together naturality diagrams:

$$\begin{aligned} \sigma \circ \tau\,(o) &= \sigma{\circ}\boldsymbol{G}_0(o) \cdot \boldsymbol{F}_1{\circ}\tau(o) = \underline{\boldsymbol{G}_0(\sigma(o)) \cdot \tau(\boldsymbol{F}_1(o))} \\ &= \boldsymbol{F}_0{\circ}\tau(o) \cdot \sigma{\circ}\boldsymbol{G}_1(o) = \tau(\boldsymbol{F}_0(o)) \cdot \boldsymbol{G}_1(\sigma(o)) \end{aligned}$$

for any object $o \in Obj(C_0)$. The alternate definitions are equal, due to the naturality of τ. In the following KIF formalism, we use the first alternative definition (underlined).

$$\begin{tikzcd}
o \arrow[d, "m"'] & \boldsymbol{G}_0(\boldsymbol{F}_0(o)) \arrow[r, "\boldsymbol{G}_0(\sigma(o))"] \arrow[d, "\boldsymbol{G}_0(\boldsymbol{F}_0(m))"] & \boldsymbol{G}_0(\boldsymbol{F}_1(o)) \arrow[r, "\tau(\boldsymbol{F}_1(o))"] \arrow[d, "\boldsymbol{G}_0(\boldsymbol{F}_1(m))"] & \boldsymbol{G}_1(\boldsymbol{F}_1(o)) \arrow[d, "\boldsymbol{G}_1(\boldsymbol{F}_2(m))"] \\
o' & \boldsymbol{G}_0(\boldsymbol{F}_0(o')) \arrow[r, "\boldsymbol{G}_0(\sigma(o'))"'] & \boldsymbol{G}_0(\boldsymbol{F}_1(o')) \arrow[r, "\tau(\boldsymbol{F}_1(o'))"'] & \boldsymbol{G}_1(\boldsymbol{F}_1(o'))
\end{tikzcd}
\qquad
\begin{tikzcd}
\boldsymbol{C}_0 \arrow[r, bend left, "\boldsymbol{F}_0"] \arrow[r, bend right, "\boldsymbol{F}_1"'] \arrow[r, phantom, "\Downarrow \sigma"] & \boldsymbol{C}_1 \arrow[r, bend left, "\boldsymbol{G}_0"] \arrow[r, bend right, "\boldsymbol{G}_1"'] \arrow[r, phantom, "\Downarrow \tau"] & \boldsymbol{C}_2
\end{tikzcd}$$

Diagram 3: Horizontal composition

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

(14) (KIF$relation horizontally-composable)
     (= (KIF$collection1 horizontally-composable) natural-transformation)
     (= (KIF$collection2 horizontal-composable) natural-transformation)
     (= (KIF$extent horizontally-composable)
        (KIF$pullback horizontally-composable-opspan))

(15) (KIF$function horizontal-composition)
     (= (KIF$source horizontal-composition)
        (KIF$pullback horizontally-composable-opspan))
     (= (KIF$target horizontal-composition) natural-transformation)
     (forall (?sigma (natural-transformation ?sigma)
              ?tau (natural-transformation ?tau) (composable ?sigma ?tau))
         (and (= (source-functor (horizontal-composition [?sigma ?tau]))
                 (FUNC$composition [(source-functor ?sigma) (source-functor ?tau)]))
              (= (target-functor (horizontal-composition [?sigma ?tau]))
                 (FUNC$composition [(target-functor ?sigma) (target-functor ?tau)]))
              (= (source-category (horizontal-composition [?sigma ?tau]))
                 (source-category ?sigma))
              (= (target-category (horizontal-composition [?sigma ?tau]))
                 (target-category ?tau))
              (= (component (horizontal-composition [?sigma ?tau]))
                 (SET.FTN$composition
                     [((SET.LIM.PBK$pairing
                           (CAT$composable-opspan (target-category ?sigma)))
                        [(SET.FTN$composition
                           [(component ?sigma) (FUNC$morphism (source-functor ?tau))])
                         (SET.FTN$composition
                           [(FUNC$object (target ?sigma)) (component ?tau)])])
                      (CAT$composition (target-category ?tau))]))))
```

- For any category $\boldsymbol{C}$ there is a *horizontal identity* natural transformation $1_{\boldsymbol{C}} : id_{\boldsymbol{C}} \Rightarrow id_{\boldsymbol{C}} : \boldsymbol{C} \rightarrow \boldsymbol{C}$ defined in terms of identity morphisms in the category $\boldsymbol{C}$: $1_{\boldsymbol{C}}\,(o) = id^{\boldsymbol{C}}_o$ for any object $o \in obj(\boldsymbol{C})$.

```
(16) (KIF$function horizontal-identity)
     (= (KIF$source horizontal-identity) CAT$category)
     (= (KIF$target horizontal-identity) natural-transformation)
     (forall (?c (CAT$category ?c))
         (and (= (source-functor (horizontal-identity ?c)) (FUNC$identity ?c))
              (= (target-functor (horizontal-identity ?c)) (FUNC$identity ?c))
              (= (component (horizontal-identity ?c)) (FUNC$identity ?c))))
```

- Given any category $\boldsymbol{C}$ and any category $\boldsymbol{J}$, the *diagonal* natural transformation $\Delta_{\boldsymbol{J},\boldsymbol{C}}$ maps a morphism $m : o_0 \rightarrow o_1$ in $\boldsymbol{C}$ to an associated constant natural transformation $\Delta_{\boldsymbol{J},\boldsymbol{C}}(m) : \Delta_{\boldsymbol{J},\boldsymbol{C}}(o_0) \Rightarrow \Delta_{\boldsymbol{J},\boldsymbol{C}}(o_1) : \boldsymbol{J} \rightarrow \boldsymbol{C}$

between constant functors – its component function maps each object $j \in obj(\mathbf{J})$ to the morphism $m \in mor(\mathbf{C})$. This constitutes the morphism part of the diagonal functor $\Delta_{\mathbf{J},\,\mathbf{C}} : \mathbf{C} \rightarrow \mathbf{C}^{\mathbf{J}}$.

```
(16) (KIF$function diagonal)
     (= (KIF$source diagonal) (KIF$power [KIF$two CAT$category]))
     (= (KIF$target diagonal) KIF$function)
     (forall (?j (CAT$category ?j) ?c (CAT$category ?c))
         (and (= (KIF$source (diagonal [?j ?c])) (CAT$morphism ?c))
              (= (KIF$target (diagonal [?j ?c])) natural-transformation)
              (forall (?m ((CAT$morphism ?c) ?m))
                  (and (= (source-category ((diagonal [?j ?c]) ?m)) ?j)
                       (= (target-category ((diagonal [?j ?c]) ?m)) ?c)
                       (= (source-functor ((diagonal [?j ?c]) ?m))
                          ((FUNC$diagonal [?j ?c]) ((CAT$source ?c) ?m)))
                       (= (target-functor ((diagonal [?j ?c]) ?m))
                          ((FUNC$diagonal [?j ?c]) ((CAT$target ?c) ?m)))))))
```

- There is a quasi-category (a foundationally large category with object and morphism collections), whose objects are functors, whose morphisms are natural transformation, whose source and target are the source and target functors for a natural transformation, whose composition is vertical composition, and whose identities are the vertical identities.
- There is a quasi-category, whose objects are categories, whose arrows are natural transformation, whose source and target are the source and target categories for a natural transformation, whose composition is horizontal composition, and whose identities are the horizontal identities.

We can prove the following theorems.

- The horizontal composition can be written in two different forms:

  $\sigma \circ \tau = (1_{\mathbf{F}0} \circ \tau) \cdot (\sigma \circ 1_{\mathbf{G}1}) = (\sigma \circ 1_{\mathbf{G}0}) \cdot (1_{\mathbf{F}1} \circ \tau)$.

- Vertical and horizontal composition satisfy the *interchange law*:

  $(\sigma_0 \cdot \sigma_1) \circ (\tau_0 \cdot \tau_1) = (\sigma_0 \circ \tau_0) \cdot (\sigma_1 \circ \tau_1)$

  for any three categories, six functors and four natural transformations as in the diagram in Figure 2.

$$\begin{tikzcd}
\mathbf{C}_0 \arrow[r, shift left=4, "\mathbf{F}_0"] \arrow[r] \arrow[r, shift right=4, "\mathbf{F}_2"'] \arrow[r, phantom, shift left=2, "\Downarrow \sigma_0"] \arrow[r, phantom, shift right=2, "\Downarrow \sigma_1"] & \mathbf{C}_1 \arrow[r, shift left=4, "\mathbf{G}_0"] \arrow[r] \arrow[r, shift right=4, "\mathbf{G}_2"'] \arrow[r, phantom, shift left=2, "\Downarrow \tau_0"] \arrow[r, phantom, shift right=2, "\Downarrow \tau_1"] & \mathbf{C}_2
\end{tikzcd}$$

Figure 2: Interchange Law

```
(forall (?sigma0 (natural-transformation ?sigma0)
         ?sigma1 (natural-transformation ?sigma1)
         ?tau0 (natural-transformation ?tau0)
         ?tau1 (natural-transformation ?tau1))
    (=> (and (vertically-composable ?sigma0 ?sigma1)
             (vertically-composable ?tau0 ?tau1)
             (horizontally-composable ?sigma0 ?tau0)
             (horizontally-composable ?sigma1 ?tau1))
        (= (horizontal-composition
               [(vertical-composition [?sigma0 ?sigma1])
                (vertical-composition [?tau0 ?tau1])])
           (vertical-composition
               [(horizontal-composition [?sigma0 ?tau0])
                (horizontal-composition [?sigma1 ?tau1])]))))
```

The Namespace of Large Adjunctions

`ADJ`

Categories are not only related by functors but also related through adjunctions. This namespace represents large adjunctions – adjunctions whose underlying and free categories are large. The content of the namespace for adjunctions is developed in chapter 4 of Mac Lane 1971. All terms declared and axiomatized in this namespace are listed in Table 1. The suggested prefix for the basic terms in this namespace is '`ADJ`', standing for adjunction: when used in an external namespace, all basic terms that originate from this namespace should be prefixed with '`ADJ`'. There is also a subcollection of terms concerned with morphisms of adjunctions. These should be prefixed with '`ADJ.MOR`'.

Table 1: Terms introduced in the Category Theory Ontology

	Collection	Function	Other

`ADJ`	`adjunction`
	`reflection`
	`coreflection`
		`underlying-category`
		`free-category`
		`left-adjoint right-adjoint`
		`unit counit`
		`monad`
		`free eilenberg-moore-comparison`
		`extension kliesli-comparison`	

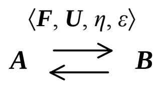
`ADJ .MOR`	`conjugate-pair`	`source target`
		`left-conjugate right-conjugate`
			`eta inst-pow`

Adjunctions

○ An *adjunction* $\langle \boldsymbol{F}, \boldsymbol{U}, \eta, \varepsilon\rangle : \boldsymbol{A} \rightarrow \boldsymbol{B}$ consists of a pair of natural transformations called the *unit* $\eta : id_{\boldsymbol{A}} \Rightarrow \boldsymbol{F} \cdot \boldsymbol{U}$ and *counit* $\varepsilon : \boldsymbol{U} \cdot \boldsymbol{F} \Rightarrow id_{\boldsymbol{B}}$ of the adjunction, a pair of functors called the *left adjoint* (or free functor) $\boldsymbol{F} : \boldsymbol{A} \rightarrow \boldsymbol{B}$ and the *right adjoint* (or underlying functor) $\boldsymbol{U} : \boldsymbol{B} \rightarrow \boldsymbol{A}$ of the adjunction, and a pair of categories called the *underlying category* $\boldsymbol{A}$ and *free category* $\boldsymbol{B}$ of the adjunction (Figure 1). An adjunction is governed by a pair of *triangle identities*,

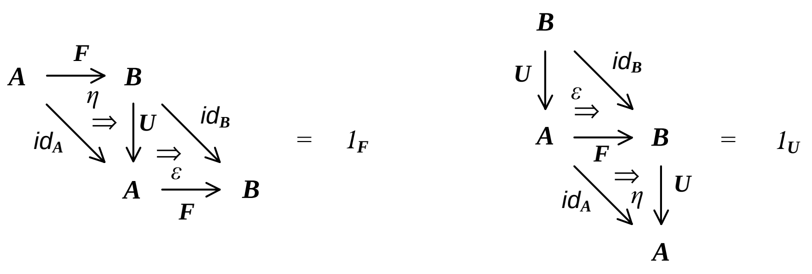
$$\eta\boldsymbol{F} \bullet \boldsymbol{F}\varepsilon = 1_{\boldsymbol{F}} \qquad \text{and} \qquad \varepsilon\boldsymbol{U} \bullet \boldsymbol{U}\eta = 1_{\boldsymbol{U}},$$

as illustrated in Diagram 1. Adjunctions are determined by the sextuple (unit, counit, left-adjoint, right-adjoint, underlying-category, free-category).



$$\begin{tikzcd} \boldsymbol{A} \arrow[r, shift left, "\langle \boldsymbol{F}, \boldsymbol{U}, \eta, \varepsilon\rangle"] & \boldsymbol{B} \arrow[l, shift left] \end{tikzcd}$$

Figure 1: Adjunction



$$\begin{tikzcd} \boldsymbol{A} \arrow[r, "\boldsymbol{F}"] \arrow[dr, "id_{\boldsymbol{A}}"'] \arrow[dr, phantom, shift left=3, "\overset{\eta}{\Rightarrow}"] & \boldsymbol{B} \arrow[d, "\boldsymbol{U}"] \arrow[dr, "id_{\boldsymbol{B}}"] \arrow[dr, phantom, shift right=3, "\overset{\varepsilon}{\Rightarrow}"'] & \\ & \boldsymbol{A} \arrow[r, "\boldsymbol{F}"'] & \boldsymbol{B} \end{tikzcd} = 1_{\boldsymbol{F}} \qquad \begin{tikzcd} \boldsymbol{B} \arrow[d, "\boldsymbol{U}"'] \arrow[dr, "id_{\boldsymbol{B}}"] \arrow[dr, phantom, shift right=3, "\overset{\varepsilon}{\Rightarrow}"'] & \\ \boldsymbol{A} \arrow[r, "\boldsymbol{F}"'] \arrow[dr, "id_{\boldsymbol{A}}"'] \arrow[dr, phantom, shift left=3, "\overset{\Rightarrow}{\eta}"] & \boldsymbol{B} \arrow[d, "\boldsymbol{U}"] \\ & \boldsymbol{A} \end{tikzcd} = 1_{\boldsymbol{U}}$$

Diagram 1: Triangle identities

```
(1) (KIF$collection adjunction)

(2) (KIF$function underlying-category)
    (= (KIF$source underlying-category) adjunction)
    (= (KIF$target underlying-category) CAT$category)

(3) (KIF$function free-category)
    (= (KIF$source free-category) adjunction)
```

```
        (= (KIF$target free-category) CAT$category)

    (4) (KIF$function left-adjoint)
        (= (KIF$source left-adjoint) adjunction)
        (= (KIF$target left-adjoint) FUNC$functor)
        (forall (?a (adjunction ?a))
            (and (= (CAT$source (left-adjoint ?a)) (underlying-category ?a))
                 (= (CAT$target (left-adjoint ?a)) (free-category ?a))))

    (5) (KIF$function right-adjoint)
        (= (KIF$source right-adjoint) adjunction)
        (= (KIF$target right-adjoint) FUNC$functor)
        (forall (?a (adjunction ?a))
            (and (= (CAT$source (right-adjoint ?a)) (free-category ?a))
                 (= (CAT$target (right-adjoint ?a)) (underlying-category ?a))))

    (6) (KIF$function unit)
        (= (KIF$source unit) adjunction)
        (= (KIF$target unit) NAT$natural-transformation)
        (forall (?a (adjunction ?a))
            (and (= (NAT$source (unit ?a))
                    (FUNC$identity (underlying-category ?a)))
                 (= (NAT$target (unit ?a))
                    (FUNC$composition [(left-adjoint ?a) (right-adjoint ?a)]))))

    (7) (KIF$function counit)
        (= (KIF$source counit) adjunction)
        (= (KIF$target counit) NAT$natural-transformation)
        (forall (?a (adjunction ?a))
            (and (= (NAT$source (counit ?a))
                    (FUNC$composition [(right-adjoint ?a) (left-adjoint ?a)]))
                 (= (NAT$target (counit ?a))
                    (FUNC$identity (free-category ?a)))))

    (8) (forall (?a (adjunction ?a))
            (= (NAT$vertical-composition
                   [(NAT$horizontal-composition
                        [(unit ?a) (NAT$vertical-identity (left-adjoint ?a))])
                    (NAT$horizontal-composition
                        [(NAT$vertical-identity (left-adjoint ?a)) (counit ?a)])])
               (NAT$vertical-identity (left-adjoint ?a))))

    (9) (forall (?a (adjunction ?a))
            (= (NAT$vertical-composition
                   [(NAT$horizontal-composition
                        [(counit ?a) (NAT$vertical-identity (right-adjoint ?a))])
                    (NAT$horizontal-composition
                        [(NAT$vertical-identity (right-adjoint ?a)) (unit ?a)])])
               (NAT$vertical-identity (right-adjoint ?a))))

    (10) (forall (?a1 (adjunction ?a1) ?a2 (adjunction ?a2))
             (=> (and (= (underlying-category ?a1) (underlying-category ?a2))
                      (= (free-category ?a1) (free-category ?a2))
                      (= (left-adjoint ?a1) (left-adjoint ?a2))
                      (= (right-adjoint ?a1) (right-adjoint ?a2))
                      (= (unit ?a1) (unit ?a2))
                      (= (counit ?a1) (counit ?a2)))
                 (= ?a1 ?a2)))
```

○ Adjunctions and universal morphisms are closely related – we can prove the following theorem: if $\boldsymbol{U} : \boldsymbol{B} \rightarrow \boldsymbol{A}$ is any functor, then $\boldsymbol{U}$ is the right adjoint functor in an adjunction $\langle \boldsymbol{F}, \boldsymbol{U}, \eta, \varepsilon \rangle : \boldsymbol{A} \rightarrow \boldsymbol{B}$ iff there is a universal morphism for every object $a \in obj(\boldsymbol{A})$.
- Given the adjunction, the universal morphism for $a \in obj(\boldsymbol{A})$ is given by $\langle \eta_a, \boldsymbol{F}(a) \rangle$.
- Given the collection of universal morphisms $\{\langle m_a, \tilde{a} \rangle \mid a \in obj(\boldsymbol{A})\}$, define the object part of $\boldsymbol{F}$ by $\boldsymbol{F}(a) = \tilde{a}$ and define the morphism part of $\boldsymbol{F}$ as follows: $\boldsymbol{F}(m : a \rightarrow a')$ is the unique morphism $\eta_a \cdot_{\boldsymbol{A}} \boldsymbol{U}(\boldsymbol{F}(m)) = m \cdot \eta_{\hat{a}}$.

We express this result in an external namespace.

```
        (forall (?u (FUNC$functor ?u))
            (<=> (exists (?adj (ADJ$adjunction ?adj))
                     (= ?u (ADJ$right-adjoint ?adj))
                 (forall (?a ((CAT$object (FUNC$target ?u)) ?a))
                     (exists (?x (((universal-morphism ?u) ?a) ?x))))))
```

○ It is a standard fact that every adjunction $\langle \boldsymbol{F}, \boldsymbol{U}, \eta, \varepsilon\rangle : \boldsymbol{A} \rightarrow \boldsymbol{B}$ gives rise to a monad $\langle \boldsymbol{T}, \eta, \mu\rangle$ in a category $\boldsymbol{A}$, where

$$\begin{aligned} \boldsymbol{T} &= \boldsymbol{F} \circ \boldsymbol{U} \\ \eta &= \eta \\ \mu &= 1_{\boldsymbol{F}} \circ \varepsilon \circ 1_{\boldsymbol{U}} \end{aligned}$$

```
(11) (KIF$function monad)
     (= (KIF$source monad) adjunction)
     (= (KIF$target monad) MND$monad)
     (forall (?a (adjunction ?a))
         (and (= (MND$category (monad ?a))
                 (underlying-category ?a))
              (= (MND$endofunctor (monad ?a))
                 (FUNC$composition [(left-adjoint ?a) (right-adjoint ?a)]))
              (= (MND$unit (monad ?a))
                 (unit ?a))
              (= (MND$multiplication (monad ?a))
                 (NAT$horizontal-composition
                     [(NAT$vertical-identity (left-adjoint ?a))
                      (NAT$horizontal-composition
                          [(counit ?a)
                           (NAT$vertical-identity (right-adjoint ?a))])]))))
```

○ Consider the opposite direction. We know that every monad $\boldsymbol{M} = \langle \boldsymbol{T}, \eta, \mu\rangle$ in the category $\boldsymbol{A}$ gives rise to two distinguished adjunctions (see the sections below that axiomatize the monad namespace),
– the Eilenberg-Moore adjunction $\langle \boldsymbol{F}^{\boldsymbol{M}}, \boldsymbol{U}^{\boldsymbol{M}}, \eta^{\boldsymbol{M}}, \varepsilon^{\boldsymbol{M}}\rangle : \boldsymbol{A}^{\boldsymbol{M}} \rightarrow \boldsymbol{A}$ and
– the Kliesli adjunction $\langle \boldsymbol{F}_{\boldsymbol{M}}, \boldsymbol{U}_{\boldsymbol{M}}, \eta_{\boldsymbol{M}}, \varepsilon_{\boldsymbol{M}}\rangle : \boldsymbol{A}_{\boldsymbol{M}} \rightarrow \boldsymbol{A}$.
If the monad $\boldsymbol{M}$ is the one generated by an adjunction $\langle \boldsymbol{F}, \boldsymbol{U}, \eta, \varepsilon\rangle : \boldsymbol{A} \rightarrow \boldsymbol{B}$, then the three adjunctions are comparable: there exists two distinguished functors, the *Kliesli comparison functor* $\boldsymbol{K}_{\boldsymbol{M}} : \boldsymbol{A}_{\boldsymbol{M}} \rightarrow \boldsymbol{B}$ and the *Eilenberg-Moore comparison functor* $\boldsymbol{K}^{\boldsymbol{M}} : \boldsymbol{B} \rightarrow \boldsymbol{A}^{\boldsymbol{M}}$, that satisfy the following identities.

$$\begin{aligned} \boldsymbol{U} &= \boldsymbol{K}^{\boldsymbol{M}} \circ \boldsymbol{U}^{\boldsymbol{M}} \\ \boldsymbol{F}^{\boldsymbol{M}} &= \boldsymbol{F} \circ \boldsymbol{K}^{\boldsymbol{M}} \\ \boldsymbol{U}_{\boldsymbol{M}} &= \boldsymbol{K}_{\boldsymbol{M}} \circ \boldsymbol{U} \\ \boldsymbol{F} &= \boldsymbol{F}_{\boldsymbol{M}} \circ \boldsymbol{K}_{\boldsymbol{M}} \end{aligned}$$

$$\begin{tikzcd}
\boldsymbol{A}_{\boldsymbol{M}} \arrow[r, "\boldsymbol{K}_{\boldsymbol{M}}"] \arrow[dr, shift left, "\boldsymbol{U}_{\boldsymbol{M}}"] & \boldsymbol{B} \arrow[r, "\boldsymbol{K}^{\boldsymbol{M}}"] \arrow[d, shift left, "\boldsymbol{U}"] & \boldsymbol{A}^{\boldsymbol{M}} \arrow[dl, shift left, "\boldsymbol{U}^{\boldsymbol{M}}"] \\
& \boldsymbol{A} \arrow[ul, shift left, "\boldsymbol{F}_{\boldsymbol{M}}"] \arrow[u, shift left, "\boldsymbol{F}"] \arrow[ur, shift left, "\boldsymbol{F}^{\boldsymbol{M}}"] &
\end{tikzcd}$$

Diagram 2: Comparison Functors

– The Eilenberg-Moore comparison functor $\boldsymbol{K}^{\boldsymbol{M}} : \boldsymbol{B} \rightarrow \boldsymbol{A}^{\boldsymbol{M}}$ maps an object $b \in obj(\boldsymbol{B})$ to the *free* algebra $\boldsymbol{K}^{\boldsymbol{M}}(b) = \langle \boldsymbol{U}(b), \boldsymbol{U}(\varepsilon(b))\rangle$ and maps a $\boldsymbol{B}$-morphism $h : b \rightarrow b'$ to the homomorphism $\boldsymbol{F}^{\boldsymbol{M}}(h) = \boldsymbol{U}(h) : \langle \boldsymbol{U}(b), \boldsymbol{U}(\varepsilon(b))\rangle \rightarrow \langle \boldsymbol{U}(b'), \boldsymbol{U}(\varepsilon(b'))\rangle$.

– The Kliesli comparison functor $\boldsymbol{K}_{\boldsymbol{M}} : \boldsymbol{A}_{\boldsymbol{M}} \rightarrow \boldsymbol{B}$ maps an object $a \in obj(\boldsymbol{A}_{\boldsymbol{M}})$ to the $\boldsymbol{B}$-object $\boldsymbol{K}^{\boldsymbol{M}}(b) = \boldsymbol{F}(a)$ and maps an $\boldsymbol{A}_{\boldsymbol{M}}$-morphism $\langle h, a'\rangle : a \rightarrow a'$, where $h : a \rightarrow \boldsymbol{T}(a')$ is an A-morphism, to the *extension* $\boldsymbol{B}$-morphism $\boldsymbol{F}(h) \cdot_{\boldsymbol{B}} \varepsilon(\boldsymbol{F}(a')) : \boldsymbol{F}(a) \rightarrow \boldsymbol{F}(a')$.

```
(12) (KIF$function free)
     (= (KIF$source free) adjunction)
     (= (KIF$target free) SET.FTN$function)
     (forall (?a (adjunction ?a))
         (and (= (SET.FTN$source (free ?a)) (CAT$object (free-category ?a)))
              (= (SET.FTN$target (free ?a)) (MND$algebra (monad ?a)))
              (= (SET.FTN$composition [(free ?a) (MND.EM$object (monad ?a))])
                 (FUNC$object (right-adjoint ?a)))
              (= (SET.FTN$composition [(free ?a) (MND.EM$structure-map (monad ?a))])
                 (SET.FTN$composition
                     [(NAT$component (counit ?a))
                      (FUNC$morphism (right-adjoint ?a))]))))

(13) (KIF$function eilenberg-moore-comparison)
     (= (KIF$source eilenberg-moore-comparison) adjunction)
     (= (KIF$target eilenberg-moore-comparison) FUNC$functor)
     (forall (?a (adjunction ?a))
         (and (= (FUNC$source (eilenberg-moore-comparison ?a)) (free-category ?a))
              (= (FUNC$target (eilenberg-moore-comparison ?a))
                 (MND.EM$eilenberg-moore (monad ?a)))
              (= (FUNC$object (eilenberg-moore-comparison ?a)) (free ?a))
              (= (SET.FTN$composition
                     [(FUNC$morphism (eilenberg-moore-comparison ?a))
                      (MND.EM$morphism (monad ?a))])
                 (FUNC$morphism (right-adjoint ?a)))))

(14) (KIF$function extension)
     (= (KIF$source extension) adjunction)
     (= (KIF$target extension) SET.FTN$function)
     (forall (?a (adjunction ?a))
         (and (= (SET.FTN$source (extension ?a))
                 (SET.LIM.PBK$pullback (MND.KLI$morphism-opspan (monad ?a))))
              (= (SET.FTN$target (extension ?a)) (CAT$morphism (free-category ?a)))
              (= (extension ?a)
                 (SET.FTN$composition
                     [((SET.LIM.PBK$pairing
                           (CAT$composable-opspan (MND$free-category ?a)))
                          [(SET.FTN$composition
                               [(SET.LIM.PBK$projection1
                                    (MND.KLI$morphism-opspan (monad ?a)))
                                (FUNC$morphism (left-adjoint ?a))])
                           (SET.FTN$composition
                               [(SET.LIM.PBK$projection2
                                    (MND.KLI$morphism-opspan (monad ?a)))
                                (SET.FTN$composition
                                    [(FUNC$object (left-adjoint ?a))
                                     (NAT$component (counit ?a))])])])
                      (CAT$composition (free-category ?a))]))))

(15) (KIF$function kliesli-comparison)
     (= (KIF$source kliesli-comparison) adjunction)
     (= (KIF$target kliesli-comparison) FUNC$functor)
     (forall (?a (adjunction ?a))
         (and (= (FUNC$source (kliesli-comparison ?a)) (MND.KLI$kliesli (monad ?a)))
              (= (FUNC$target (kliesli-comparison ?a)) (free-category ?a))
              (= (FUNC$object (kliesli-comparison ?a))
                 (FUNC$object (left-adjoint ?a)))
              (= (FUNC$morphism (kliesli-comparison ?a)) (extension ?a))))
```

○ Given any two categories ***A*** and ***B***, a (*strong*) *reflection* of ***A*** into ***B*** is an adjunction $\langle \boldsymbol{F}, \boldsymbol{U}, \eta, 1_{id\boldsymbol{B}} \rangle : \boldsymbol{A} \rightarrow \boldsymbol{B}$ whose counit is the identity, $\varepsilon = 1_{id\boldsymbol{B}}$, with $\boldsymbol{U} \cdot \boldsymbol{F} = id_{\boldsymbol{B}}$, so that ***U*** has injective object and morphism functions and ***B*** is a subcategory of ***A*** via ***U***. That is, a subcategory $\boldsymbol{B} \subseteq \boldsymbol{A}$ is a *reflection* of ***A*** when the injection functor has a left adjoint right inverse (lari). Dually, given any two categories ***A*** and ***B***, a (*strong*) *coreflection* of ***A*** into ***B*** is an adjunction $\langle \boldsymbol{F}, \boldsymbol{U}, 1_{id\boldsymbol{A}}, \varepsilon \rangle : \boldsymbol{A} \rightarrow \boldsymbol{B}$ whose unit is the identity, $\eta = 1_{id\boldsymbol{A}}$, with $\boldsymbol{F} \cdot \boldsymbol{U} = id_{\boldsymbol{A}}$, so that ***F*** has injective object and morphism functions and ***A*** is a subcategory of ***B*** via ***F***. That is, a subcategory $\boldsymbol{A} \subseteq \boldsymbol{B}$ is a *coreflection* of ***B*** when the injection functor has a right adjoint right inverse (rari).

```
    (16) (KIF$collection reflection)
         (KIF$subcollection reflection adjunction)
         (forall (?a (adjunction ?a))
             (<=> (reflection ?a)
                  (and (= (FUNC$composition [(right-adjoint ?a) (left-adjoint ?a)])
                          (FUNC$identity (free-category ?a)))
                       (= (counit ?a)
                          (NAT$vertical-identity
                              (FUNC$identity (free-category ?a)))))))
```

```
(17) (KIF$collection coreflection)
     (KIF$subcollection coreflection adjunction)
     (forall (?a (adjunction ?a))
         (<=> (coreflection ?a)
              (and (= (FUNC$composition [(left-adjoint ?a) (right-adjoint ?a)])
                      (FUNC$identity (underlying-category ?a)))
                   (= (unit ?a)
                      (NAT$vertical-identity
                          (FUNC$identity (underlying-category ?a)))))))
```

Adjunction Morphisms

ADJ.MOR

Adjunctions are related (vertically) by conjugate pairs of natural transformations.

○ Suppose that two adjunctions $\langle \boldsymbol{F}, \boldsymbol{U}, \eta, \varepsilon\rangle, \langle \boldsymbol{F}', \boldsymbol{U}', \eta', \varepsilon'\rangle : A \to B$ share a common underlying (source) category $\boldsymbol{A}$ and a common free (target) category $\boldsymbol{B}$. A *conjugate pair* of natural transformations $\langle\sigma, \tau\rangle : \langle \boldsymbol{F}, \boldsymbol{U}, \eta, \varepsilon\rangle \Rightarrow \langle \boldsymbol{F}', \boldsymbol{U}', \eta', \varepsilon'\rangle : \boldsymbol{A} \to \boldsymbol{B}$ from source adjunction $\langle \boldsymbol{F}, \boldsymbol{U}, \eta, \varepsilon\rangle$ to target functor $\langle \boldsymbol{F}', \boldsymbol{U}', \eta', \varepsilon'\rangle$, visualized 2-dimensionally in Figure 2, consists of a *left conjugate* natural transformation $\sigma : \boldsymbol{F} \Rightarrow \boldsymbol{F}'$ between the left adjoint (free) functors and a (contravariant) *right conjugate* natural transformation $\tau : \boldsymbol{U}' \Rightarrow \boldsymbol{U}$ between the right adjoint (underlying) functors, which satisfy either of the equivalent conditions in Table 2:

$$\begin{tikzcd}
\boldsymbol{A} \arrow[r, shift left=2, "\langle \boldsymbol{F}, \boldsymbol{U}, \eta, \varepsilon\rangle"] \arrow[r, shift right=2, swap, "\langle \boldsymbol{F}', \boldsymbol{U}', \eta', \varepsilon'\rangle"] \arrow[r, phantom, "\Downarrow\langle\sigma,\tau\rangle"] & \boldsymbol{B}
\end{tikzcd}$$

Figure 2: Conjugate Pair

Table 2: Equivalent Conditions for Conjugate Pairs

$$\begin{aligned}
\tau &= \boldsymbol{U}'\eta \bullet \boldsymbol{U}'\sigma\boldsymbol{U} \bullet \varepsilon'\boldsymbol{U} & \qquad \sigma &= \eta'\boldsymbol{F} \bullet \boldsymbol{F}'\tau\boldsymbol{F} \bullet \boldsymbol{F}'\varepsilon \\
\tau\boldsymbol{F} \bullet \varepsilon &= \boldsymbol{U}'\sigma \bullet \varepsilon' & \qquad \eta \bullet \sigma\boldsymbol{U} &= \eta' \bullet \boldsymbol{F}'\tau
\end{aligned}$$

As these equivalents indicate, the natural transformation σ determines the natural transformation τ, and vice versa. Therefore, a conjugate pair is determined by either its left or right conjugate natural transformation. The left two equivalent conditions in Table 1 are used here.

```
(11) (KIF$collection conjugate-pair)

(12) (KIF$function source)
     (= (KIF$source source) conjugate-pair)
     (= (KIF$target source) ADJ$adjunction)

(13) (KIF$function target)
     (= (KIF$source target) conjugate-pair)
     (= (KIF$target target) ADJ$adjunction)

(14) (forall (?p (conjugate-pair ?p))
         (and (= (ADJ$underlying-category (source ?p))
                 (ADJ$underlying-category (target ?p)))
              (= (ADJ$free-category (source ?p))
                 (ADJ$free-category (target ?p)))))

(15) (KIF$function left-conjugate)
     (= (KIF$source left-conjugate) conjugate-pair)
     (= (KIF$target left-conjugate) NAT$natural-transformation)
     (forall (?p (conjugate-pair ?p))
         (and (= (NAT$source (left-conjugate ?a))
                 (ADJ$left-adjoint (source ?p)))
              (= (NAT$target (left-conjugate ?a))
                 (ADJ$left-adjoint (target ?p)))))

(16) (KIF$function right-conjugate)
     (= (KIF$source right-conjugate) conjugate-pair)
     (= (KIF$target right-conjugate) NAT$natural-transformation)
     (forall (?p (conjugate-pair ?p))
         (and (= (NAT$source (right-conjugate ?a))
                 (ADJ$right-adjoint (target ?p)))
```

```
                  (= (NAT$target (right-conjugate ?a))
                     (ADJ$right-adjoint (source ?p)))))

    (17) (forall (?p (conjugate-pair ?p))
             (and [τ = U′η • U′σU • ε′U]
                  (= (left-conjugate ?a)
                     (ADJ$vertical-composition
                       [(ADJ$vertical-composition
                          [(ADJ$horizontal-composition
                             (NAT$vertical-identity (ADJ$right-adjoint (target ?p)))
                             (ADJ$unit (source ?p))])
                           (ADJ$horizontal-composition
                             [(ADJ$horizontal-composition
                                [(NAT$vertical-identity (ADJ$right-adjoint (target ?p)))
                                 (left-conjugate ?p)])
                              (NAT$vertical-identity (ADJ$right-adjoint (source ?p)))])])
                        (ADJ$horizontal-composition
                          [(ADJ$counit (target ?p))
                           (NAT$vertical-identity (ADJ$right-adjoint (source ?p)))])]))
                  [τF • ε = U′σ • ε′]
                  (= (ADJ$vertical-composition
                       [(ADJ$horizontal-composition
                          [(right-conjugate ?p)
                           (NAT$vertical-identity (ADJ$left-adjoint (source ?p)))])
                        (ADJ$counit (source ?p))])
                     (ADJ$vertical-composition
                       [(ADJ$horizontal-composition
                          [(NAT$vertical-identity (ADJ$right-adjoint (target ?p)))
                           (left-conjugate ?p)])
                        (ADJ$counit (target ?p))]))
         ))
```

Examples

Here are examples of adjunctions defined elsewhere, but asserted to be adjunctions here.

○ There is a natural transformation η from the identity functor on Classification to the composition of the underlying instance and instance power functors, whose component at any classification is the extent infomorphism associated with that classification. The underlying instance functor

  inst : Classification → $\mathrm{Set}^{\mathrm{op}}$

  is left adjoint *inst* ⊣ *pow* to the instance power functor

  pow : $\mathrm{Set}^{\mathrm{op}}$ → Classification,

  and η is the unit of this adjunction. Here is the KIF formalization for these facts, expressed in an external namespace.

```
(NAT$natural-transformation eta)
(= (NAT$source eta) (FUNC$identity Classification))
(= (NAT$target eta) (FUNC$composition [instance instance-power]))
(= (NAT$component eta) cls$extent)

(FUNC$composable instance instance-power)
(= (FUNC$composition [instance-power instance]) (FUNC$identity Set))

(ADJ$adjunction inst-pow)
(= (ADJ$underlying-category inst-pow) Classification)
(= (ADJ$free-category inst-pow)       Set)
(= (ADJ$left-adjoint inst-pow)        instance)
(= (ADJ$right-adjoint inst-pow)       instance-power)
(= (ADJ$unit inst-pow)                eta)
(= (ADJ$counit inst-pow)              (NAT$identity (FUNC$identity Set)))
```

The Namespace of Large Monads

`MND`

In one sense, monads are universal algebra lifted to category theory. This namespace represents large monads – monads whose category and functor are large. Monads are also known as “triples” in the literature. The content of the namespace for monads is developed in chapter 6 of Mac Lane 1971. All terms declared and axiomatized in this namespace are listed in Table 1. The suggested prefix for the basic terms in this namespace is ‘`MND`’, standing for monad: when used in an external namespace, all basic terms that originate from this namespace should be prefixed with ‘`MND`’. In addition, there are two specialized collections of terms – either centered around the Eilenberg-Moore category of algebras or centered around the Kliesli category of terms. The suggested prefixes for these terms are ‘`MND.EM`’ and ‘`MND.KLI`’, respectively.

Table 1: Terms introduced in the Category Theory Ontology

	Collection	Function	Other

`MND`	`monad`	`category functor`
		`unit multiplication`	

	`monad-morphism`	`source target`
		`natural-transformation`	

`MND`
`.EM`
		`algebra`
		`object structure-map`
		`homomorphism`
		`source target`
		`morphism`
		`composable-opspan composable`
		`composition identity`
		`eilenberg-moore`
		`underlying free`
		`unit counit adjunction`	

`MND`
`.KLI`
		`morphism-opspan`
		`identity-cone`
		`composable-opspan`
		`extension`
		`kliesli`
		`underlying`
		`embed free`
		`unit counit adjunction`	

Monads and Monad Morphisms

For any type of algebra (such as groups, complete semilattices, etcetra) there is a category of algebras Alg, its (right adjoint) forgetful functor U from Alg to Set and its (left adjoint) free functor F in the reverse direction. The composite functor $T = F \circ U$ on Set comes equipped with two natural transformations that give it a monoid-like structure.

$$\begin{tikzcd}
\boldsymbol{T}^3 \arrow[r, "id_{\boldsymbol{T}} \circ \mu"] \arrow[d, "\mu \circ id_{\boldsymbol{T}}"'] & \boldsymbol{T}^2 \arrow[d, "\mu"] \\
\boldsymbol{T}^2 \arrow[r, "\mu"'] & \boldsymbol{T}
\end{tikzcd}
\qquad
\begin{tikzcd}
\boldsymbol{T} = id_{\boldsymbol{A}} \circ \boldsymbol{T} \arrow[r, "\eta \circ id_{\boldsymbol{T}}"] \arrow[dr, "id_{\boldsymbol{T}}"'] & \boldsymbol{T}^2 \arrow[d, "\mu"] & \boldsymbol{T} \circ id_{\boldsymbol{A}} = \boldsymbol{T} \arrow[l, "id_{\boldsymbol{T}} \circ \eta"'] \arrow[dl, "id_{\boldsymbol{T}}"] \\
& \boldsymbol{T} &
\end{tikzcd}$$

Diagram 1: Monad

○ A *monad* $\boldsymbol{M} = \langle \boldsymbol{T}, \eta, \mu \rangle$ on a category $\boldsymbol{A}$ is a triple consisting of an underlying endofunctor $\boldsymbol{T} : \boldsymbol{A} \rightarrow \boldsymbol{A}$ and two natural transformations

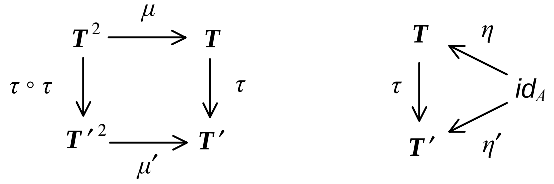
$$\eta : id_{\boldsymbol{A}} \Rightarrow \boldsymbol{T} \text{ and } \mu : \boldsymbol{T} \circ \boldsymbol{T} \Rightarrow \boldsymbol{T}$$

which satisfy the commuting diagrams in Diagram 1 (where $\boldsymbol{T}^2 = \boldsymbol{T} \circ \boldsymbol{T}$ and $\boldsymbol{T}^3 = \boldsymbol{T} \circ \boldsymbol{T} \circ \boldsymbol{T}$).

The natural transformation $\eta : id_A \Rightarrow \boldsymbol{T}$ is called the *unit* of the monad, and the natural transformation $\mu : \boldsymbol{T} \circ \boldsymbol{T} \Rightarrow \boldsymbol{T}$ is called the *multiplication* of the monad. In Diagram 1, the axiom on the left is called the *associative law* for the monad and the axioms on the right are called the *left unit law* and the *right unit law*, respectively. Monads are determined by their (**`functor`**, unit, multiplication) triples.

```
(1) (KIF$collection monad)

(2) (KIF$function category)
    (= (KIF$source category) monad)
    (= (KIF$target category) CAT$category)

(3) (KIF$function functor)
    (= (KIF$source functor) monad)
    (= (KIF$target functor) FUNC$functor)
    (forall (?m (monad ?m))
        (and (= (FUNC$source (functor ?m)) (category ?m))
             (= (FUNC$target (functor ?m)) (category ?m))))

(4) (KIF$function unit)
    (= (KIF$source unit) monad)
    (= (KIF$target unit) NAT$natural-transformation)
    (forall (?m (monad ?m))
        (and (= (NAT$source-functor (unit ?m)) (FUNC$identity (category ?m)))
             (= (NAT$target-functor (unit ?m)) (functor ?m))))

(5) (KIF$function multiplication)
    (= (KIF$source multiplication) monad)
    (= (KIF$target multiplication) NAT$natural-transformation)
    (forall (?m (monad ?m))
        (and (= (NAT$source-functor (multiplication ?m))
                (FUNC$composition [(functor ?m) (functor ?m)]))
             (= (NAT$target-functor (multiplication ?m)) (functor ?m))))

(6) (forall (?m (monad ?m))
        (= (NAT$vertical-identity
               [(NAT$horizontal-composition
                    [(NAT$vertical-identity (functor ?m)) (multiplication ?m)])
                (multiplication ?m)])
           (NAT$vertical-identity
               [(NAT$horizontal-composition
                    [(multiplication ?m) (NAT$vertical-identity (functor ?m))])
                (multiplication ?m)])))

(7) (forall (?m (monad ?m))
        (and (= (NAT$vertical-identity
                    [(NAT$horizontal-composition
                         [(unit ?m) (NAT$vertical-identity (functor ?m))])
                     (multiplication ?m)])
                (NAT$vertical-identity (functor ?m)))
             (= (NAT$vertical-identity
                    [(NAT$horizontal-composition
                         [(NAT$vertical-identity (functor ?m)) (unit ?m)])
                     (multiplication ?m)])
                (NAT$vertical-identity (functor ?m)))))

(8) (forall (?m1 (monad ?m1) ?m2 (monad ?m2))
         (=> (and (= (functor ?m1) (functor ?m2))
                  (= (unit ?m1) (unit ?m2))
                  (= (multiplication ?m1) (multiplication ?m2)))
             (= ?m1 ?m2)))
```

- Monads are related by their morphisms. A *morphism of monads* $\tau : \langle \boldsymbol{T}, \eta, \mu \rangle \Rightarrow \langle \boldsymbol{T}', \eta', \mu' \rangle$ is a natural transformation $\tau : \boldsymbol{T} \Rightarrow \boldsymbol{T}'$ which preserves multiplication and unit in the sense that the diagrams in Table 2 commute. In Diagram 2, the axiom on the left represents *preservation of multiplication* and the



$$\begin{tikzcd} \boldsymbol{T}^2 \arrow[r, "\mu"] \arrow[d, "\tau \circ \tau"'] & \boldsymbol{T} \arrow[d, "\tau"] \\ \boldsymbol{T}'^2 \arrow[r, "\mu'"'] & \boldsymbol{T}' \end{tikzcd} \qquad \begin{tikzcd} \boldsymbol{T} \arrow[d, "\tau"'] & \\ \boldsymbol{T}' & id_A \arrow[ul, "\eta"'] \arrow[l, "\eta'"] \end{tikzcd}$$

Diagram 2: Monad Morphism

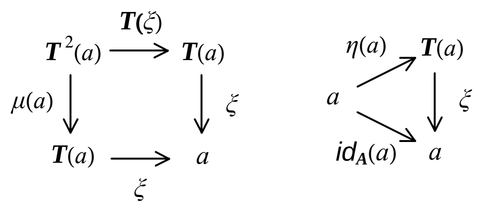
axiom on the right represents *preservation of unit*.

```
(9) (KIF$collection monad-morphism)

(10) (KIF$function source)
     (= (KIF$source source) monad-morphism)
     (= (KIF$target source) monad)

(11) (KIF$function target)
     (= (KIF$source target) monad-morphism)
     (= (KIF$target target) monad)

(12) (KIF$function natural-transformation)
     (= (KIF$source natural-transformation) monad-morphism)
     (= (KIF$target natural-transformation) NAT$natural-transformation)
     (forall (?t (monad-morphism ?t))
         (and (= (category (source ?t)) (category (target ?t)))
              (= (NAT$source-functor (natural-transformation ?t))
                 (functor (source ?t)))
              (= (NAT$target-functor (natural-transformation ?t))
                 (functor (target ?t)))))
              (= (NAT$vertical-composition [(multiplication (source ?t)) ?t])
                 (NAT$vertical-composition
                     [(NAT$horizontal-composition [?t ?t])
                      (multiplication (target ?t))]))
              (= (NAT$vertical-composition [(unit (source ?t)) ?t])
                 (unit (target ?t)))))
```

Eilenberg-Moore

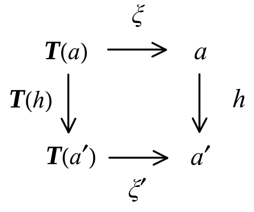
`MND.EM`

For any monad $\boldsymbol{M} = \langle \boldsymbol{T}, \eta, \mu \rangle$, the Eilenberg-Moore category of algebras represents universal algebras and their homomorphisms.



$$\begin{tikzcd}
\boldsymbol{T}^2(a) \arrow[r, "\boldsymbol{T}(\xi)"] \arrow[d, "\mu(a)"'] & \boldsymbol{T}(a) \arrow[d, "\xi"] \\
\boldsymbol{T}(a) \arrow[r, "\xi"'] & a
\end{tikzcd}
\qquad
\begin{tikzcd}
 & \boldsymbol{T}(a) \arrow[d, "\xi"] \\
a \arrow[ur, "\eta(a)"] \arrow[r, "id_{\boldsymbol{A}}(a)"'] & a
\end{tikzcd}$$

Diagram 3: Algebra



$$\begin{tikzcd}
\boldsymbol{T}(a) \arrow[r, "\xi"] \arrow[d, "\boldsymbol{T}(h)"'] & a \arrow[d, "h"] \\
\boldsymbol{T}(a') \arrow[r, "\xi'"'] & a'
\end{tikzcd}$$

Diagram 4: Homomorphism

○ If $\boldsymbol{M} = \langle \boldsymbol{T}, \eta, \mu \rangle$ is a monad on category $\boldsymbol{A}$, then an $\boldsymbol{M}$-*algebra* $\langle a, \xi \rangle$ is a pair consisting of an *object* $a \in$ *obj*($\boldsymbol{A}$), and a morphism $\xi : \boldsymbol{T}(a) \rightarrow a$ (called the *structure map* of the algebra) which makes the diagrams in Diagram 3 commute. The diagram on the left in Diagram 3 is called the *associative law* for the algebra and the diagram on the right is called the *unit law*.

```
(1) (KIF$function algebra)
    (= (KIF$source algebra) MND$monad)
    (= (KIF$target algebra) SET$class)

(2) (KIF$function object)
    (= (KIF$source object) MND$monad)
    (= (KIF$target object) SET.FTN$function)
    (forall (?m (MND$monad ?m))
        (and (= (SET.FTN$source (object ?m)) (algebra ?m))
             (= (SET.FTN$target (object ?m)) (CAT$object (MND$category ?m)))))

(3) (KIF$function structure-map)
    (= (KIF$source structure-map) MND$monad)
    (= (KIF$target structure-map) SET.FTN$function)
    (forall (?m (MND$monad ?m))
        (and (= (SET.FTN$source (structure-map ?m)) (algebra ?m))
             (= (SET.FTN$target (structure-map ?m)) (CAT$morphism (MND$category ?m)))
             (= (SET.FTN$composition
                    [(structure-map ?m) (CAT$source (MND$category ?m))])
                (SET.FTN$composition
                    [(object ?m) (FUNC$object (MND$functor ?m))]))
             (= (SET.FTN$composition
                    [(structure-map ?m) (CAT$target (MND$category ?m))])
                (object ?m))
             (forall (?a ((algebra ?m) ?a))
                 (and (= ((CAT$composition (MND$category ?m))
                             [(FUNC$morphism (MND$functor ?m))
                                  ((structure-map ?m) ?a))
                              ((structure-map ?m) ?a)])
                         ((CAT$composition (MND$category ?m))
                             [((NAT$component (MND$multiplication ?m))
                                  ((object ?m) ?a))
                              ((structure-map ?m) ?a)]))
                      (= ((CAT$composition (MND$category ?m))
                             [((NAT$component (MND$unit ?m))
                                  ((object ?m) ?a))
                              ((structure-map ?m) ?a)])
                         ((CAT$identity (MND$category ?m))
                             ((object ?m) ?a)))))))
```

○ If $\boldsymbol{M} = \langle \boldsymbol{T}, \eta, \mu \rangle$ is a monad on category $\boldsymbol{A}$, an $\boldsymbol{M}$-*homomorphism* $\boldsymbol{h} : \langle a, \xi \rangle \rightarrow \langle a', \xi' \rangle$ is an $\boldsymbol{A}$-morphism $h : a \rightarrow a'$ between the underlying objects that preserves the algebraic structure by satisfying the commutative diagram in Diagram 4.

```
(4) (KIF$function homomorphism)
    (= (KIF$source homomorphism) MND$monad)
    (= (KIF$target homomorphism) SET.class)

(5) (KIF$function source)
    (= (KIF$source source) MND$monad)
    (= (KIF$target source) SET.FTN$function)
    (forall (?m (MND$monad ?m))
        (and (= (SET.FTN$source (source ?m)) (homomorphism ?m))
             (= (SET.FTN$target (source ?m)) (algebra ?m))))

(6) (KIF$function target)
    (= (KIF$source target) MND$monad)
    (= (KIF$target target) SET.FTN$function)
    (forall (?m (MND$monad ?m))
        (and (= (SET.FTN$source (target ?m)) (homomorphism ?m))
             (= (SET.FTN$target (target ?m)) (algebra ?m))))

(7) (KIF$function morphism)
    (= (KIF$source morphism) MND$monad)
    (= (KIF$target morphism) SET.FTN$function)
    (forall (?m (MND$monad ?m))
        (and (= (SET.FTN$source (morphism ?m)) (homomorphism ?m))
             (= (SET.FTN$target (morphism ?m)) (CAT$morphism (MND$category ?m)))
             (= (SET.FTN$composition
                    [(morphism ?m) (CAT$source (MND$category ?m))])
                (SET.FTN$composition [(source ?m) (object ?m)]))
             (= (SET.FTN$composition
                    [(morphism ?m) (CAT$target (MND$category ?m))])
                (SET.FTN$composition [(target ?m) (object ?m)]))
             (forall (?h ((homomorphism ?m) ?h))
                 (= ((CAT$composition (MND$category ?m))
                        [((structure-map ?m) (source ?h)) ?h])
                    ((CAT$composition (MND$category ?m))
                        [((FUNC$morphism (MND$functor ?m)) ?h)
                         ((structure-map ?m) (target ?h))])))))
```

o For any monad $\boldsymbol{M}$, two $\boldsymbol{M}$-homomorphisms functions are *composable* when the target of the first is equal to the source of the second. The *composition* of two composable homomorphisms $h_1 : \langle a, \xi \rangle \rightarrow \langle a', \xi' \rangle$ and $h_2 : \langle a', \xi' \rangle \rightarrow \langle a'', \xi'' \rangle$ is define in terms of the composition of the underlying morphisms $h_1 \cdot_{\boldsymbol{A}} h_2 : a \rightarrow a''$.

```
(8) (KIF$function composable-opspan)
    (= (KIF$source composable-opspan) MND$monad)
    (= (KIF$target composable-opspan) SET.LIM.PBK$diagram)
    (forall (?m (MND$monad ?m))
        (and (= (SET.LIM.PRD2$class1 (composable-opspan ?m)) (homomorphism ?m))
             (= (SET.LIM.PRD2$class2 (composable-opspan ?m)) (homomorphism ?m))
             (= (SET.LIM.PRD2$opvertex (composable-opspan ?m)) (algebra ?m))
             (= (SET.LIM.PRD2$opfirst (composable-opspan ?m)) (target ?m))
             (= (SET.LIM.PRD2$opsecond (composable-opspan ?m)) (source ?m))))

(9) (KIF$function composable)
    (= (KIF$source composable) MND$monad)
    (= (KIF$target composable) REL$relation)
    (forall (?m (MND$monad ?m))
        (= (composable ?m) (SET.LIM.PBK$relation (composable-opspan ?m))))

(10) (KIF$function composition)
     (= (KIF$source composition) MND$monad)
     (= (KIF$target composition) SET.FTN$function)
     (forall (?m (MND$monad ?m))
         (and (= (SET.FTN$source (composition ?m)) (REL$extent (composable ?m)))
              (= (SET.FTN$target (composition ?m)) (homomorphism ?m))
              (forall (?h1 ((homomorphism ?m) ?h1)
                       ?h2 ((homomorphism ?m) ?h2) ((composable ?m) ?h1 ?h2))
                  (= (morphism ((composition ?m) [?h1 ?h2]))
                     ((CAT$composition (MND$category ?m))
                         [(morphism ?h1) (morphism ?h2)])))))
```

- For any $\boldsymbol{M}$-algebra $\langle a, \xi\rangle$ there is an *identity* $\boldsymbol{M}$-homomorphism $id_{\boldsymbol{M}}(\langle a, \xi\rangle) : \langle a, \xi\rangle \rightarrow \langle a, \xi\rangle$.

```
(11) (KIF$function identity)
     (= (KIF$source identity) MND$monad)
     (= (KIF$target identity) SET.FTN$function)
     (forall (?m (MND$monad ?m))
         (and (= (SET.FTN$source (identity ?m)) (algebra ?m))
              (= (SET.FTN$target (identity ?m)) (homomorphism ?m))
              (forall (?a ((algebra ?m) ?a))
                  (= (morphism ((identity ?m) ?a))
                     ((CAT$identity (MND$category ?m))
                         ((object ?m) ?a))))))
```

- For any monad $\boldsymbol{M} = \langle \boldsymbol{T}, \eta, \mu\rangle$ on category $\boldsymbol{A}$, the $\boldsymbol{M}$-algebras and $\boldsymbol{M}$-homomorphisms form the *Eilenberg-Moore category* $\boldsymbol{A}^{\boldsymbol{M}}$.

```
(12) (KIF$function eilenberg-moore)
     (= (KIF$source eilenberg-moore) MND$monad)
     (= (KIF$target eilenberg-moore) CAT$category)
     (forall (?m (monad ?m))
         (and (= (CAT$object (eilenberg-moore ?m)) (algebra ?m))
              (= (CAT$morphism (eilenberg-moore ?m)) (homomorphism ?m))
              (= (CAT$source (eilenberg-moore ?m)) (source ?m))
              (= (CAT$target (eilenberg-moore ?m)) (target ?m))
              (= (CAT$composition (eilenberg-moore ?m)) (composition ?m))
              (= (CAT$identity (eilenberg-moore ?m)) (identity ?m))))
```

- For any monad $\boldsymbol{M} = \langle \boldsymbol{T}, \eta, \mu\rangle$ on category $\boldsymbol{A}$, there is
  - a (right adjoint) *underlying* functor $\boldsymbol{U}^{\boldsymbol{M}} : \boldsymbol{A}^{\boldsymbol{M}} \rightarrow \boldsymbol{A}$,
  - a (left adjoint) *free* functor $\boldsymbol{F}^{\boldsymbol{M}} : \boldsymbol{A} \rightarrow \boldsymbol{A}^{\boldsymbol{M}}$ that maps an object $a \in obj(\boldsymbol{A})$ to the free algebra $\langle \boldsymbol{T}(a), \mu(a)\rangle$ and maps an $\boldsymbol{A}$-morphism $h : a \rightarrow a'$ to the $\boldsymbol{M}$-homomorphism $\boldsymbol{F}^{\boldsymbol{M}}(h) = \boldsymbol{T}(h) : \langle \boldsymbol{T}(a), \mu(a)\rangle \rightarrow \langle \boldsymbol{T}(a'), \mu(a')\rangle$,
  - a *unit* natural transformation $\eta^{\boldsymbol{M}} : id_{\boldsymbol{A}} \Rightarrow \boldsymbol{F}^{\boldsymbol{M}} \circ \boldsymbol{U}^{\boldsymbol{M}}$ with component $\eta^{\boldsymbol{M}}(a) = \eta(a)$ at any object $a \in obj(\boldsymbol{A})$, and
  - a *counit* natural transformation $\varepsilon^{\boldsymbol{M}} : \boldsymbol{U}^{\boldsymbol{M}} \circ \boldsymbol{F}^{\boldsymbol{M}} \Rightarrow id_{\boldsymbol{A}}$ with component $\varepsilon^{\boldsymbol{M}}(\langle a, \xi\rangle) = \xi$ at any algebra $\langle a, \xi\rangle$.

  This data forms an *adjunction* $\langle \boldsymbol{F}^{\boldsymbol{M}}, \boldsymbol{U}^{\boldsymbol{M}}, \eta^{\boldsymbol{M}}, \varepsilon^{\boldsymbol{M}}\rangle : \boldsymbol{A} \rightarrow \boldsymbol{A}^{\boldsymbol{M}}$.

```
(13) (KIF$function underlying)
     (= (KIF$source underlying) MND$monad)
     (= (KIF$target underlying) FUNC$functor)
```

```
        (forall (?m (MND$monad ?m))
            (and (= (FUNC$source (underlying ?m)) (eilenberg-moore ?m))
                 (= (FUNC$target (underlying ?m)) (MND$category ?m))
                 (= (FUNC$object (underlying ?m)) (object ?m))
                 (= (FUNC$morphism (underlying ?m)) (morphism ?m))))

   (14) (KIF$function free)
        (= (KIF$source free) MND$monad)
        (= (KIF$target free) FUNC$functor)
        (forall (?m (MND$monad ?m))
            (and (= (FUNC$source (free ?m)) (MND$category ?m))
                 (= (FUNC$target (free ?m)) (eilenberg-moore ?m))
                 (= (SET.FTN$composition [(FUNC$object (free ?m)) (object ?m)])
                    (FUNC$object (MND$functor ?m)))
                 (= (SET.FTN$composition [(FUNC$object (free ?m)) (structure-map ?m)])
                    (NAT$component (MND$multiplication ?m)))
                 (= (SET.FTN$composition [(FUNC$morphism (free ?m)) (morphism ?m)])
                    (FUNC$morphism (MND$functor ?m)))))

   (15) (KIF$function unit)
        (= (KIF$source unit) MND$monad)
        (= (KIF$target unit) NAT$natural-transformation)
        (forall (?m (MND$monad ?m))
            (and (= (NAT$source-functor (unit ?m))
                    (FUNC$identity (MND$category ?m)))
                 (= (NAT$target-functor (unit ?m))
                    (FUNC$composition [(free ?m) (underlying ?m)]))
                 (= (NAT$component (unit ?m)) (NAT$component (MND$unit ?m)))))

   (16) (KIF$function counit)
        (= (KIF$source counit) MND$monad)
        (= (KIF$target counit) NAT$natural-transformation)
        (forall (?m (MND$monad ?m))
            (and (= (NAT$source-functor (counit ?m))
                    (FUNC$composition [(underlying ?m) (free ?m)]))
                 (= (NAT$target-functor (counit ?m))
                    (FUNC$identity (MND$category ?m)))
                 (= (NAT$component (counit ?m)) (structure-map ?m))))

   (17) (KIF$function adjunction)
        (= (KIF$source adjunction) MND$monad)
        (= (KIF$target adjunction) ADJ$adjunction)
        (forall (?m (MND$monad ?m))
            (and (= (ADJ$left-adjoint (adjunction ?m)) (free ?m))
                 (= (ADJ$right-adjoint (adjunction ?m)) (underlying ?m))
                 (= (ADJ$unit (adjunction ?m)) (unit ?m))
                 (= (ADJ$counit (adjunction ?m)) (counit ?m))))
```

○ We can then prove the theorem that the monad generated by the Eilenberg-Moore adjunction is the original monad. We state this in an external namespace.

```
        (forall (?m (MND$monad ?m))
            (= (ADJ$monad (MND.EM$adjunction ?m)) ?m))
```

Kliesli

`MND.KLI`

For any monad $\boldsymbol{M} = \langle \boldsymbol{T}, \eta, \mu \rangle$, the *Kliesli category* represents the free part of universal algebras.

○ Let $\boldsymbol{M} = \langle \boldsymbol{T}, \eta, \mu \rangle$ be a monad on category $\boldsymbol{A}$. Restrict your attention to the morphisms in $\boldsymbol{A}$ of the form $h : a \rightarrow \boldsymbol{T}(a')$. The Kliesli category $\boldsymbol{A_M}$ has this as a morphism with source object $a \in obj(\boldsymbol{A_M})$ and target object $a' \in obj(\boldsymbol{A_M})$. So $obj(\boldsymbol{A_M}) = obj(\boldsymbol{A})$, $mor(\boldsymbol{A_M}) \subseteq mor(\boldsymbol{A})$, composition of two morphisms $h : a \rightarrow \boldsymbol{T}(a')$ and $h' : a' \rightarrow \boldsymbol{T}(a'')$ is defined by $h \cdot_{\boldsymbol{AM}} h' = h \cdot_{\boldsymbol{A}} h'^{\#}$ where $h'^{\#} = \boldsymbol{T}(h') \cdot_{\boldsymbol{A}} \mu(a'')$ is the *extension* of h (here $(\text{-})^{\#}$ is the extension operator), and the identity at an object $a \in obj(\boldsymbol{A_M})$ is the unit component $\eta(a) : a \rightarrow \boldsymbol{T}(a)$. More precisely, as can be seen below, a morphism is a pair $\langle h, a' \rangle$, where $h : a \rightarrow \boldsymbol{T}(a')$ is an $\boldsymbol{A}$-morphism. Clearly, foundational pullback opspans, cocones and pairing play a key role in the definition of the Kliesli category.

```
(1) (KIF$function morphism-opspan)
    (= (KIF$source morphism-opspan) MND$monad)
    (= (KIF$target morphism-opspan) SET.LIM.PBK$opspan)
    (forall (?m (MND$monad ?m))
        (and (= (SET.LIM.PBK$class1 (morphism-opspan ?m))
                (CAT$morphism (MND$category ?m)))
             (= (SET.LIM.PBK$class2 (morphism-opspan ?m))
                (CAT$object (MND$category ?m)))
             (= (SET.LIM.PBK$opvertex (morphism-opspan ?m))
                (CAT$object (MND$category ?m)))
             (= (SET.LIM.PBK$opfirst (morphism-opspan ?m))
                (CAT$target (MND$category ?m)))
             (= (SET.LIM.PBK$opsecond (morphism-opspan ?m))
                (FUNC$object (MND$functor ?m)))))

(2) (KIF$function identity-cone)
    (= (KIF$source identity-cone) MND$monad)
    (= (KIF$target identity-cone) SET.LIM.PBK$cocone)
    (forall (?m (MND$monad ?m))
        (and (= (SET.LIM.PBK$cone-diagram (identity-cone ?m))
                (morphism-opspan ?m))
             (= (SET.LIM.PBK$vertex (identity-cone ?m))
                (CAT$object (MND$category ?m)))
             (= (SET.LIM.PBK$first (identity-cone ?m))
                (NAT$component (MND$unit ?m)))
             (= (SET.LIM.PBK$second (identity-cone ?m))
                (SET.FTN$identity (MND$category ?m)))))

(3) (KIF$function composable-opspan)
    (= (KIF$source composable-opspan) MND$monad)
    (= (KIF$target composable-opspan) SET.LIM.PBK$opspan)
    (forall (?m (MND$monad ?m))
        (and (= (SET.LIM.PBK$class1 (composable-opspan ?m))
                (SET.LIM.PBK$pullback (morphism-opspan ?m)))
             (= (SET.LIM.PBK$class2 (composable-opspan ?m))
                (SET.LIM.PBK$pullback (morphism-opspan ?m)))
             (= (SET.LIM.PBK$opvertex (composable-opspan ?m))
                (CAT$object (MND$category ?m)))
             (= (SET.LIM.PBK$opfirst (composable-opspan ?m))
                (SET.LIM.PBK$projection2 (morphism-opspan ?m)))
             (= (SET.LIM.PBK$opsecond (composable-opspan ?m))
                (SET.FTN$composition
                    [(SET.LIM.PBK$projection1 (morphism-opspan ?m))
                     (CAT$source (MND$category ?m))]))))

(4) (KIF$function extension)
    (= (KIF$source extension) MND$monad)
    (= (KIF$target extension) SET.FTN$function)
    (forall (?m (MND$monad ?m))
        (and (= (SET.FTN$source (extension ?m))
                (SET.LIM.PBK$pullback (morphism-opspan ?m)))
             (= (SET.FTN$target (extension ?m))
                (CAT$morphism (MND$category ?m)))
             (= (extension ?m)
                (SET.FTN$composition
                    [((SET.LIM.PBK$pairing (CAT$composable-opspan (MND$category ?m)))
                         [(SET.FTN$composition
                              [(SET.LIM.PBK$projection1 (morphism-opspan ?m))
                               (FUNC$morphism (MND$functor ?m))])
                          (SET.FTN$composition
                              [(SET.LIM.PBK$projection2 (morphism-opspan ?m))
                               (NAT$component (MND$multiplication ?m))])])
                     (CAT$composition (MND$category ?m))]))))

(5) (KIF$function kliesli)
    (= (KIF$source kliesli) MND$monad)
    (= (KIF$target kliesli) CAT$category)
    (forall (?m (monad ?m))
        (and (= (CAT$object (kliesli ?m))
                (CAT$object (MND$category ?m)))
```

```
                  (= ((CAT$morphism (kliesli ?m))
                     (SET.LIM.PBK$pullback (morphism-opspan ?m)))
                  (= (CAT$source (kliesli ?m))
                     (SET.FTN$composition
                         [(SET.LIM.PBK$projection1 (morphism-opspan ?m))
                          (CAT$source (MND$category ?m))]))
                  (= (CAT$target (kliesli ?m))
                     (SET.LIM.PBK$projection2 (morphism-opspan ?m)))
                  (= (CAT$composable (kliesli ?m))
                     (SET.LIM.PBK$relation (composable-opspan ?m)))
                  (= (SET.FTN$composition
                         [(CAT$composition (kliesli ?m))
                          (SET.LIM.PBK$projection1 (composable-opspan ?m))])
                     (SET.FTN$composition
                         [((SET.LIM.PBK$pairing (CAT$composable-opspan (MND$category ?m)))
                              [(SET.FTN$composition
                                   [(SET.LIM.PBK$projection1 (composable-opspan ?m))
                                    (SET.LIM.PBK$projection1 (morphism-opspan ?m))])
                               (SET.FTN$composition
                                   [(SET.LIM.PBK$projection2 (composable-opspan ?m))
                                    (extension ?m)])])
                          (CAT$composition (MND$category ?m))]))
                  (= (CAT$identity (kliesli ?m))
                     (SET.LIM.PBK$mediator (identity-cone ?m)))))
```

- For any monad $\boldsymbol{M} = \langle \boldsymbol{T}, \eta, \mu \rangle$ on category $\boldsymbol{A}$, there is
  - an *underlying* functor $\boldsymbol{U_M} : \boldsymbol{A_M} \rightarrow \boldsymbol{A}$ that maps an object $a \in obj(\boldsymbol{A_M})$ to the object $\boldsymbol{T}(a) \in obj(\boldsymbol{A})$ and maps a morphism $\langle h, a' \rangle : a \rightarrow a'$ in $\boldsymbol{A_M}$ to the extension morphism $h^{\#} : \boldsymbol{T}(a) \rightarrow \boldsymbol{T}(a')$ in $\boldsymbol{A}$,
  - a *free* functor $\boldsymbol{F_M} : \boldsymbol{A} \rightarrow \boldsymbol{A_M}$ that is the identity on objects and maps a $\boldsymbol{A}$-morphism $h : a \rightarrow a'$ to the *embedded* morphism $\langle h \cdot \eta(a'), a' \rangle : a \rightarrow a'$ in $\boldsymbol{A_M}$,
  - a *unit* natural transformation $\eta_{\boldsymbol{M}} : id_{\boldsymbol{A}} \Rightarrow \boldsymbol{F_M} \circ \boldsymbol{U_M}$ with component $\eta_{\boldsymbol{M}}(a) = \eta(a)$ at any object $a \in obj(\boldsymbol{A})$, and
  - a *counit* natural transformation $\varepsilon_{\boldsymbol{M}} : \boldsymbol{U_M} \circ \boldsymbol{F_M} \Rightarrow id_{\boldsymbol{AM}}$ with component $\varepsilon_{\boldsymbol{M}}(a) = \langle id_{\boldsymbol{A}}(\boldsymbol{T}(a)), a \rangle : \boldsymbol{T}(a) \rightarrow a$ at any $a \in obj(\boldsymbol{A_M})$.

  This data forms an *adjunction* $\langle \boldsymbol{F_M}, \boldsymbol{U_M}, \eta_{\boldsymbol{M}}, \varepsilon_{\boldsymbol{M}} \rangle : \boldsymbol{A} \rightarrow \boldsymbol{A_M}$.

```
(6) (KIF$function underlying)
    (= (KIF$source underlying) MND$monad)
    (= (KIF$target underlying) FUNC$functor)
    (forall (?m (MND$monad ?m))
        (and (= (FUNC$source (underlying ?m)) (kliesli ?m))
             (= (FUNC$target (underlying ?m)) (MND$category ?m))
             (= (FUNC$object (underlying ?m)) (FUNC$object (MND$functor ?m)))
             (= (FUNC$morphism (underlying ?m)) (extension ?m))))

(7) (KIF$function embed)
    (= (KIF$source embed) MND$monad)
    (= (KIF$target embed) SET.FTN$function)
    (forall (?m (MND$monad ?m))
        (and (= (SET.FTN$source (embed ?m)) (CAT$morphism (MND$category ?m)))
             (= (SET.FTN$target (embed ?m)) (CAT$morphism (kliesli ?m)))
             (= (SET.FTN$composition
                    [(embed ?m) (SET.LIM.PBK$projection1 (morphism-opspan ?m))])
                (SET.FTN$composition
                    [((SET.LIM.PBK$pairing (CAT$composable-opspan (MND$category ?m)))
                         [(SET.FTN$identity (CAT$morphism (MND$category ?m)))
                          (SET.FTN$composition
                              [(CAT$target (MND$category ?m))
                               (NAT$component (MND$unit ?m))])])
                     (CAT$composition (MND$category ?m))]))
             (= (SET.FTN$composition
                    [(embed ?m) (SET.LIM.PBK$projection2 (morphism-opspan ?m))])
                (CAT$target (MND$category ?m)))))

(8) (KIF$function free)
    (= (KIF$source free) MND$monad)
    (= (KIF$target free) FUNC$functor)
```

```
        (forall (?m (MND$monad ?m))
            (and (= (FUNC$source (free ?m)) (MND$category ?m))
                 (= (FUNC$target (free ?m)) (kliesli ?m))
                 (= (FUNC$object (free ?m))
                    (SET.FTN$identity (CAT$object (MND$category ?m))))
                 (= (FUNC$morphism (free ?m)) (embed ?m))))

    (9) (KIF$function unit)
        (= (KIF$source unit) MND$monad)
        (= (KIF$target unit) NAT$natural-transformation)
        (forall (?m (MND$monad ?m))
            (and (= (NAT$source-functor (unit ?m)) (FUNC$identity (MND$category ?m)))
                 (= (NAT$target-functor (unit ?m))
                    (FUNC$composition [(free ?m) (underlying ?m)]))
                 (= (NAT$component (unit ?m)) (NAT$component (MND$unit ?m)))))

    (10) (KIF$function counit)
         (= (KIF$source counit) MND$monad)
         (= (KIF$target counit) NAT$natural-transformation)
         (forall (?m (MND$monad ?m))
             (and (= (NAT$source-functor (counit ?m))
                     (FUNC$composition [(underlying ?m) (free ?m)]))
                  (= (NAT$target-functor (counit ?m)) (FUNC$identity (MND$category ?m)))
                  (= (NAT$component (counit ?m))
                     (SET.FTN$composition
                         [(FUNC$object (MND$functor ?m))
                          (CAT$identity (MND$category ?m))]))))

    (11) (KIF$function adjunction)
         (= (KIF$source adjunction) MND$monad)
         (= (KIF$target adjunction) NAT$natural-transformation)
         (forall (?m (MND$monad ?m))
             (and (= (NAT$underlying-functor (adjunction ?m)) (underlying ?m))
                  (= (NAT$free-functor (adjunction ?m)) (free ?m))
                  (= (NAT$unit (adjunction ?m)) (unit ?m))
                  (= (NAT$counit (adjunction ?m)) (counit ?m))))
```

○ We can then prove the theorem that the monad generated by the Kliesli adjunction is the original monad. We state this theorem in an external namespace.

```
         (forall (?m (MND$monad ?m))
             (= (ADJ$monad (MND.KLI$adjunction ?m)) ?m))
```

The Namespace of Colimits/Limits

`COL`

The Information Flow Framework proposes to relate ontologies via morphisms and to compose them using colimits. This section provides the abstract colimit foundation for that proposal. Colimits are important for manipulating and composing ontologies expressed in the object language. The use of colimits advocates a "building blocks approach" for ontology construction. Continuing the metaphor, this approach understands that the mortar between the ontological blocks must be strong and resilient in order to adequately support the ontological building, and requests that methods for composing component ontologies, such as merging, mapping and aligning ontologies, be made very explicit so that they can be analyzed. The Specware system of the Kestrel Institute, which is based on category theory, supports creation and combination of specifications (ontology analogs) using colimits. A compact but detailed discussion of classifications and infomorphisms with applications to this building blocks approach for ontology construction is given in the 6th ISKO paper The Information Flow Foundation for Conceptual Knowledge Organization.

This namespace represents large colimits – colimits for diagrams with a large shape category. Colimits form a separate namespace in the Category Theory Ontology. The content of the namespace for colimits is developed in chapters 3 and 5 of Mac Lane 1971. The suggested prefix for the basic terms in this namespace is '`COL`', standing for colimit: when used in an external namespace, all basic terms that originate from this namespace should be prefixed with '`COL`'. The basic colimit terms are terms for general colimits and colimit fibers. In addition, within the colimit namespace there are several specialized collections of terms concerned with primitive colimits, and thus forming subnamespaces: initial objects, binary coproducts, coequalizers and pushouts. The suggested prefixes for these subnamespaces are '`COL.INIT`', '`COL.PRD2`', '`COL.COEQ`' and '`MCOL.PSH`', respectively. To completely express colimits, we need elements from all the basic components of category theory – categories, functors, natural transformations and adjunctions. As such, colimits provide a glimpse of the other parts of the Category Theory Ontology – the other basic components used in colimits are indicated by the namespace prefixes in the KIF formalism.

Table 1: Terms introduced in the Category Theory Ontology

	Collections	Function	Other

`COL`	`small-cocomplete`	`diagram shape`
		`cocone cocone-diagram opvertex`
		`colimiting-cocone colimit`
		`comediator`
		`cocomplete`	

		`diagram-fiber`
		`cocone-fiber cocone-diagram-fiber opvertex-fiber`
		`colimiting-cocone-fiber colimit-fiber`
		`comediator-fiber`	

`COL .INIT`		`diagram = empty`
		`cocone opvertex`
		`colimit = initial`
		`counique`	

`COL .COPRD2`		`diagram = pair object1 object2`
		`opposite`
		`cocone cocone-diagram opvertex opfirst opsecond`
		`colimiting-cocone colimit = binary-coproduct`
		`injection1 injection2`
		`comediator`	

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

`COL .PSH`		`diagram = span object 1 object2 vertex first second`
		`pair opposite coequalizer-diagram = parallel-pair`
		`cocone cocone-diagram opvertex opfirst opsecond`
		`colimiting-cocone colimit = pushout`
		`injection1 injection2`
		`comediator`	

General Colimits

A given a category $\boldsymbol{C}$ serves as an environment or mathematical context within which colimits can be constructed. Most of the concepts below are parameterized by a contextual category.

○ Given a category $\boldsymbol{C}$, a *diagram* $\boldsymbol{D}$ in $\boldsymbol{C}$ is a functor $\boldsymbol{D} : \boldsymbol{J} \rightarrow \boldsymbol{C}$ from some shape or indexing category $\boldsymbol{J}$ to the base category $\boldsymbol{C}$. In the KIF formalism below the *shape* of a diagram in $\boldsymbol{C}$ is defined as its source category. In applications a diagram is defined to be a graph morphism $\boldsymbol{D} : \boldsymbol{G} \Rightarrow |\boldsymbol{C}|$ from some or indexing graph $\boldsymbol{G}$ to the underlying graph of the base category $\boldsymbol{C}$. However, this is equivalent to a functor $\boldsymbol{D}^{\#} : \mathit{free}(\boldsymbol{G}) \Rightarrow \boldsymbol{C}$ from $\boldsymbol{J} = \mathit{free}(\boldsymbol{G})$ the free (path) category over the shape graph to the base category C – special case of the previous functorial version.

```
(1) (KIF$function diagram)
    (= (KIF$source diagram) CAT$category)
    (= (KIF$target diagram) KIF$collection)
    (forall (?c (CAT$category ?c))
        (and (KIF$subcollection (diagram ?c) FUNC$functor)
             (forall (?d (FUNC$functor ?d))
                 (<=> ((diagram ?c) ?d)
                      (= (FUNC$target ?d) ?c)))))

(2) (KIF$function shape)
    (= (KIF$source shape) CAT$category)
    (= (KIF$target shape) KIF$function)
    (forall (?c (CAT$category ?c))
        (and (= (KIF$source (shape ?c)) (diagram ?c))
             (= (KIF$target (shape ?c)) CAT$category)
             (forall (?d ((diagram ?c) ?d))
                 (= ((shape ?c) ?d) (FUNC$source ?d)))))
```

○ Given a category $\boldsymbol{C}$ a *cocone* $\tau : \boldsymbol{D} \Rightarrow \Delta_{\boldsymbol{J},\boldsymbol{C}}(o) : \boldsymbol{J} \rightarrow \boldsymbol{C}$ (Diagram 1) is a natural transformation from a diagram $\boldsymbol{D}$ in the category $\boldsymbol{C}$ of some shape $\boldsymbol{J}$ to the constant functor $\Delta_{\boldsymbol{J},\boldsymbol{C}}(o)$ for some object $o \in \mathit{obj}(\boldsymbol{C})$. The source functor $\boldsymbol{D}$ is called the *base* of the cocone τ. The object $o \in \mathit{obj}(\boldsymbol{C})$ is called the *opvertex* of the cocone τ. A cocone is analogous to the upper bound of a subset of a partial order – the order is analogous to the category $\boldsymbol{C}$, the chosen subset of preorder elements is analogous to the base diagram $\boldsymbol{D}$, and the upper bound element is analogous to the opvertex $\boldsymbol{C}$-object o. The opvertex function restricts cocones to be natural transformations whose target is a constant functor.

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

Diagram 1: Cocone

```
(3) (KIF$function cocone)
    (= (KIF$source cocone) CAT$category)
    (= (KIF$target cocone) KIF$collection)
    (forall (?c (CAT$category ?c))
        (and (KIF$subcollection (cocone ?c) NAT$natural-transformation)
             (forall (?t ((cocone ?c) ?t))
                 (= (NAT$target-category ?t) ?c))))

(4)  (KIF$function cocone-diagram)
     (KIF$function base)
     (= base cocone-diagram)
     (= (KIF$source cocone-diagram) CAT$category)
     (= (KIF$target cocone-diagram) KIF$function)
     (forall (?c (CAT$category ?c))
         (and (= (KIF$source (cocone-diagram ?c)) (cocone ?c))
              (= (KIF$target (cocone-diagram ?c)) (diagram ?c))
              (forall (?t ((cocone ?c) ?t))
                  (= ((cocone-diagram ?c) ?t)
                     (NAT$source-functor ?t)))))

(5)  (KIF$function opvertex)
     (= (KIF$source opvertex) CAT$category)
     (= (KIF$target opvertex) KIF$function)
     (forall (?c (CAT$category ?c))
         (and (= (KIF$source (opvertex ?c)) (cocone ?c))
              (= (KIF$target (opvertex ?c)) (CAT$object ?c))
```

```
                  (forall (?t ((cocone ?c) ?t))
                      (= ((FUNC$diagonal [(NAT$source-category ?t) ?c]) ((opvertex ?c) ?t)))
                         (NAT$target-functor ?t))))
```

○ Of course, the shape of the base of a cocone is its source category.

```
         (forall (?c (CAT$category ?c) ?t ((cocone ?c) ?t))
             (= (shape ((base ?c) ?t)) (NAT$source-category ?t)))
```

○ Colimits are the opvertices of special cocones. Given a category $\boldsymbol{C}$ a *colimiting-cocone* in $\boldsymbol{C}$ is a universal cocone (Diagram 2) – it consists of a cocone from a diagram $\boldsymbol{D}$ in $\boldsymbol{C}$ of some shape $\boldsymbol{J}$ to an opvertex $\bar{o} \in obj(\boldsymbol{C})$

$$\gamma : \boldsymbol{D} \Rightarrow \Delta_{\boldsymbol{C},\boldsymbol{J}}(\bar{o}) : \boldsymbol{J} \to \boldsymbol{C}$$

that is *universal*: for any other cocone

$$\tau : \boldsymbol{D} \Rightarrow \Delta_{\boldsymbol{C},\boldsymbol{J}}(o) : \boldsymbol{J} \to \boldsymbol{C}$$

with the same base diagram, there is a unique comediating morphism $m : \bar{o} \to o$ with $\gamma(j) \cdot m = \tau(j)$ for any indexing object $j \in obj(\boldsymbol{J})$, or equivalently as natural transformations, with $\gamma \bullet \Delta_{\boldsymbol{J},\boldsymbol{C}}(m) = \tau$. The *opvertex* $\bar{o} = \sum\boldsymbol{D}$ of a colimiting-cocone γ is called a *colimit*. Let $colim_{\boldsymbol{C}}(\boldsymbol{D})$ denote the function that takes a colimiting cocone to its opvertex, an object of $\boldsymbol{C}$. This will be the empty function, if no colimits exist for diagram $\boldsymbol{D}$. In the same way that a cocone is analogous to the upper bound, a colimit is analogous to a least upper bound (or supremum) of a subset of a partial order.

$$\begin{tikzcd}
j \arrow[d, "n"'] & \boldsymbol{D}(j) \arrow[r, "\gamma(j)"] \arrow[d, "\boldsymbol{D}(n)"'] \arrow[rd, "\tau(j)"] & \bar{o} \arrow[d, "m"] \\
j' & \boldsymbol{D}(j') \arrow[r, "\tau(j')"'] \arrow[ru, "\gamma(j')"] & o
\end{tikzcd}$$

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

Diagram 2: Colimiting Cocone

```
(6) (KIF$function colimiting-cocone)
    (= (KIF$source colimiting-cocone) CAT$category)
    (= (KIF$target colimiting-cocone) KIF$function)
    (forall (?c (CAT$category ?c))
        (and (= (KIF$source (colimiting-cocone ?c)) (diagram ?c))
             (= (KIF$target (colimiting-cocone ?c)) KIF$collection)
             (forall (?d ((diagram ?c) ?d))
                 (and (KIF$subcollection ((colimiting-cocone ?c) ?d) (cocone ?c))
                      (forall (?g (((colimiting-cocone ?c) ?d) ?g))
                          (= ?d ((cocone-diagram ?c) ?g))))))

(7)  (KIF$function colimit)
     (= (KIF$source colimit) CAT$category)
     (= (KIF$target colimit) KIF$function)
     (forall (?c (CAT$category ?c))
         (and (= (KIF$source (colimit ?c)) (diagram ?c))
              (= (KIF$target (colimit ?c)) KIF$function)
              (forall (?d ((diagram ?c) ?d))
                  (and (= (KIF$source ((colimit ?c) ?d)) ((colimiting-cocone ?c) ?d))
                       (= (KIF$target ((colimit ?c) ?d)) (CAT$object ?c))
                       (forall (?g (((colimiting-cocone ?c) ?d) ?g))
                           (= (((colimit ?c) ?d) ?g) ((opvertex ?c) ?g)))))))
```

○ For any context(ual category) $\boldsymbol{C}$ and any cocone τ with a base diagram $\boldsymbol{D}$, there is KIF *comediator* function from the collection of colimiting cocones with base $\boldsymbol{D}$ to the class of $\boldsymbol{C}$-morphisms (Figure 6): the comediator of a colimiting cocone γ is a $\boldsymbol{C}$-morphism whose source is the colimit of γ and whose target is the opvertex of τ. This is the unique function that commutes with the colimiting cocone γ and the given cocone τ. We use a KIF definite description abbreviation to define this. Existence and uniqueness represents the universality of the colimit.

```
(8) (KIF$function comediator)
    (= (KIF$source comediator) CAT$category)
    (= (KIF$target comediator) KIF$function)
    (forall (?c (CAT$category ?c))
      (and (= (KIF$source (comediator ?c)) (cocone ?c))
           (= (KIF$target (comediator ?c)) KIF$function)
           (forall (?t ((cocone ?c) ?t))
             (and (= (KIF$source ((comediator ?c) ?t))
                     ((colimiting-cocone ?c) ((cocone-diagram ?c) ?t)))
                  (= (KIF$target ((comediator ?c) ?t)) (CAT$morphism ?c))
```

```
                        (forall (?g (((colimiting-cocone ?c) ((cocone-diagram ?c) ?t)) ?g))
                          (= (((comediator ?c) ?t) ?g)
                             (the (?m (CAT$morphism ?c) ?m))
                               (and (= ((CAT$source ?c) ?m)
                                       (((colimit ?c) ((cocone-diagram ?c) ?t)) ?g))
                                    (= ((CAT$target ?c) ?m)
                                       ((opvertex ?c) ?t))
                                    (= (NAT$vertical-composition
                                           [?g ((NAT$diagonal
                                                 [((shape ?c) ((cocone-diagram ?c) ?t)) ?c]) ?m)])
                                       ?t)))))))))
```

- The collection of colimiting cocones may be empty – it is empty if no colimits for the diagram ***D*** exist in the category ***C***. If it is nonempty for any diagram of shape ***J***, then ***C*** is said to have ***J***-*colimits* and to be ***J***-*cocomplete*. A category ***C*** is *small cocomplete*, if it has colimits for any small diagram of ***C***; that is, when it is ***J***-cocomplete for all small shape categories ***J***. This is a concrete concept, since it uses concepts in the lower metalevel.

```
(9) (KIF$function cocomplete)
    (= (KIF$source cocomplete) CAT$category)
    (= (KIF$target cocomplete) KIF$collection)
    (forall (?j (CAT$category ?j))
        (and (KIF$subcollection (cocomplete ?j) CAT$category)
             (forall (?c (CAT$category ?c))
                 (<=> ((cocomplete ?j) ?c)
                      (forall (?d ((diagram ?c) ?d))
                          (=> (= ((shape ?c) ?d) ?j)
                              (exists (?g (((colimiting-cocone ?c) ?d) ?g)))))))))

(10) (KIF$collection small-cocomplete)
     (KIF$subcollection small-cocomplete CAT$category)
     (forall (?c (CAT$category ?c))
         (<=> (small-cocomplete ?c)
              (forall (?j (CAT$small ?j))
                  ((cocomplete ?j) ?c))))
```

- It is a standard theorem that given a category ***C*** and a diagram ***D*** in the category ***C***, any two colimits are isomorphic. The following KIF expresses this in an external namespace.

```
     (forall (?c (CAT$category ?c) ?d ((diagram ?c) ?d)
              ?g1 (((colimiting-cocone ?c) ?d) ?g1)
              ?g2 (((colimiting-cocone ?c) ?d) ?g2))
         ((CAT$isomorphic ?c) (((colimit ?c) ?d) ?g1) (((colimit ?c) ?d) ?g2)))
```

$$\begin{tikzcd}
 & \text{Object}_{\boldsymbol{C}} & & \\
 & & \boldsymbol{D} \arrow[d, phantom, sloped, "\in"] & \boldsymbol{J} \quad \boldsymbol{C} \arrow[d, phantom, "\in\ \in"] \\
\text{Colim-Cocone}_{\boldsymbol{C},\boldsymbol{D}} \arrow[uur, "colim_{\boldsymbol{C},\boldsymbol{D}}"] \arrow[r, phantom, "\subseteq"] & \text{Cocone}_{\boldsymbol{C}} \arrow[uu, "opvertex_{\boldsymbol{C}}"'] \arrow[r, "base_{\boldsymbol{C}}"] \arrow[d, phantom, sloped, "\supseteq"] & \text{Diagram}_{\boldsymbol{C}} \arrow[r, "shape_{\boldsymbol{C}}"] \arrow[d, phantom, sloped, "\supseteq"] & \text{Category} \\
 & \text{Nat-Trans} \arrow[r, "src\text{-}cat"] & \text{Functor} &
\end{tikzcd}$$

Diagram 3: Core Colimit Collections and Functions

Shape Fibers

- For any fixed diagram shape ***J*** we name the *fiber* of all *diagrams* in category ***C*** with that shape. The diagram collection over ***C*** is partitioned into the ***J***-th diagram fibers, as the shape ***J*** ranges over all categories.

```
(11) (KIF$function diagram-fiber)
     (= (KIF$source diagram-fiber) CAT$category)
     (= (KIF$target diagram-fiber) KIF$function)
     (forall (?c (CAT$category ?c))
         (and (= (KIF$source (diagram-fiber ?c)) CAT$category)
              (= (KIF$target (diagram-fiber ?c)) KIF$collection)
              (forall (?j (CAT$category ?j))
                  (and (KIF$subcollection ((diagram-fiber ?c) ?j) (diagram ?c))
                       (= ((diagram-fiber ?c) ?j) (FUNC$exponent [?j ?c]))
                       (forall (?d ((diagram ?c) ?d))
                           (<=> (((diagram-fiber ?c) ?j) ?d)
                                (= ((shape ?c) ?d) ?j)))))))
```

○ For any fixed diagram shape ***J*** we name the subcollection of the cocone collection – the *fiber* of all *cocones* in contextual category ***C*** indexed by ***J*** (whose base diagram has shape ***J***). The cocone collection over ***C*** is partitioned into the ***J***-th cocone fibers, as the base is in the ***J***-th diagram fiber (has shape ***J***) as ***J*** ranges over all categories. The base (cocone-diagram) fiber function is defined as the restriction to the ***J***-th fiber cocone collection at its source and the ***J***-th fiber diagram collection at its target, as ***J*** ranges over all categories. The opvertex fiber function is defined as the restriction to the ***J***-th fiber cocone collection at its source, as ***J*** ranges over all categories.

```
(12) (KIF$function cocone-fiber)
     (= (KIF$source cocone-fiber) CAT$category)
     (= (KIF$target cocone-fiber) KIF$function)
     (forall (?c (CAT$category ?c))
         (and (= (KIF$source (cocone-fiber ?c)) CAT$category)
              (= (KIF$target (cocone-fiber ?c)) KIF$collection)
              (forall (?j (CAT$category ?j))
                  (and (KIF$subcollection ((cocone-fiber ?c) ?j) (cocone ?c))
                       (forall (?t ((cocone ?c) ?t))
                           (<=> (((cocone-fiber ?c) ?j) ?t)
                                (((diagram-fiber ?c) ?j) ((base ?c) ?t))))))))

(13) (KIF$function cocone-diagram-fiber)
     (KIF$function base-fiber)
     (= base-fiber cocone-diagram-fiber)
     (= (KIF$source cocone-diagram-fiber) CAT$category)
     (= (KIF$target cocone-diagram-fiber) KIF$function)
     (forall (?c (CAT$category ?c))
         (and (= (KIF$source (cocone-diagram-fiber ?c)) CAT$category)
              (= (KIF$target (cocone-diagram-fiber ?c)) KIF$function)
              (forall (?j (CAT$category ?j))
                  (and (= (KIF$source ((cocone-diagram-fiber ?c) ?j))
                          ((cocone-fiber ?c) ?j)))
                       (= (KIF$target ((cocone-diagram-fiber ?c) ?j))
                          ((diagram-fiber ?c) ?j))
                       (KIF$restriction
                           ((cocone-diagram-fiber ?c) ?j) (cocone-diagram ?c))))))

(14) (KIF$function opvertex-fiber)
     (= (KIF$source opvertex-fiber) CAT$category)
     (= (KIF$target opvertex-fiber) KIF$function)
     (forall (?c (CAT$category ?c))
         (and (= (KIF$source (opvertex-fiber ?c)) CAT$category)
              (= (KIF$target (opvertex-fiber ?c)) KIF$function)
              (forall (?j (CAT$category ?j))
                  (and (= (KIF$source ((opvertex-fiber ?c) ?j)) ((cocone-fiber ?c) ?j))
                       (= (KIF$target ((opvertex-fiber ?c) ?j)) (CAT$object ?c))
                       (KIF$restriction ((opvertex-fiber ?c) ?j) (opvertex ?c))))))
```

○ For any fixed diagram shape ***J*** we name the *fiber* of all *colimiting cocones* of a diagram of shape ***J*** in contextual category ***C***. The colimiting cocone collection of diagram ***D*** is partitioned into the ***J***-th colimiting cocone fibers, as the base shape ***J*** ranges over all categories. The *colimit* fiber function is defined as the restriction to the ***J***-th fiber colimiting cocone collection of ***D*** at its source, as ***J*** ranges over all categories.

```
(15) (KIF$function colimiting-cocone-fiber)
     (= (KIF$source colimiting-cocone-fiber) CAT$category)
     (= (KIF$target colimiting-cocone-fiber) KIF$function)
     (forall (?c (CAT$category ?c))
         (and (= (KIF$source (colimiting-cocone-fiber ?c)) CAT$category)
              (= (KIF$target (colimiting-cocone-fiber ?c)) KIF$function)
              (forall (?j (CAT$category ?j))
                  (and (= (KIF$source ((colimiting-cocone-fiber ?c) ?j))
                           ((diagram-fiber ?c) ?j))
                       (= (KIF$target ((colimiting-cocone-fiber ?c) ?j))
                           KIF$collection)
                       (forall (?d (((diagram-fiber ?c) ?j) ?d))
                           (= (((colimiting-cocone-fiber ?c) ?j) ?d)
                              (KIF$binary-intersection
                                  [((cocone ?c) ?j) ((colimiting-cocone ?c) ?d)])))))))

(16)  (KIF$function colimit-fiber)
      (= (KIF$source colimit-fiber) CAT$category)
      (= (KIF$target colimit-fiber) KIF$function)
      (forall (?c (CAT$category ?c))
          (and (= (KIF$source (colimit-fiber ?c)) CAT$category)
               (= (KIF$target (colimit-fiber ?c)) KIF$function)
               (forall (?j (CAT$category ?j))
                   (and (= (KIF$source ((colimit-fiber ?c) ?j))
                           ((diagram-fiber ?c) ?j))
                        (= (KIF$target ((colimit-fiber ?c) ?j)) KIF$function)
                        (forall (?d (((diagram-fiber ?c) ?j) ?d))
                            (and (= (KIF$source (((colimit-fiber ?c) ?j) ?d))
                                    (((colimiting-cocone-fiber ?c) ?j) ?d))
                                 (= (KIF$target (((colimit-fiber ?c) ?j) ?d))
                                    (CAT$object ?c))
                                    (KIF$restriction
                                        (((colimit-fiber ?c) ?j) ?d)
                                        ((colimit ?c) ?d)))))))))
```

- The *comediator* fiber function is defined as the restriction to the ***J***-th fiber at its source, as ***J*** ranges over all categories.

```
(17) (KIF$function comediator-fiber)
     (= (KIF$source comediator-fiber) CAT$category)
     (= (KIF$target comediator-fiber) KIF$function)
     (forall (?c (CAT$category ?c))
         (and (= (KIF$source (comediator-fiber ?c)) CAT$category)
              (= (KIF$target (comediator-fiber ?c)) KIF$function)
              (forall (?j (CAT$category ?j))
                  (and (= (KIF$source ((comediator-fiber ?c) ?j)) ((cocone-fiber ?c) ?j))
                       (= (KIF$target ((comediator-fiber ?c) ?j)) KIF$function)
                       (forall (?t (((cocone-fiber ?c) ?j) ?t))
                           (and (= (KIF$source (((comediator-fiber ?c) ?j) ?t))
                                   (((colimiting-cocone-fiber ?c) ?j)
                                    ((cocone-diagram ?c) ?t)))
                                (= (KIF$target (((comediator-fiber ?c) ?j) ?t))
                                   (CAT$morphism ?c))
                                (KIF$restriction
                                    (((comediator-fiber ?c) ?j) ?t)
                                    ((comediator ?c) ?t))))))))
```

Finite Colimits

The general KIF formulation for colimits can be related to several special kinds of finite colimits: initial objects, binary coproducts, coequalizers and pushouts. For convenience of representation it is common to use special terminology for a few particular kinds of colimits: initial objects, binary coproducts, coequalizers and pushouts. Finite colimits are colimits that can be expressed as composites of these four primitives. There is a common formalism that encodes these primitives. This commonality has been expressed in the generalized colimits discussed above. Each primitive colimit is typed by the shape of its diagrams: initial objects are primitive colimits whose diagram shape is the *empty* category, binary coproducts are primitive colimits whose diagram shape is the discrete category *two*, coequalizers are primitive colimits whose diagram shape is the category *parallel pair*, and pushouts are primitive colimits whose diagram shape is the category *span*. All of these are specified through colimit fibers.

Initial Objects

COL.INIT

Given a category ***C***, an *initial object* in ***C*** (Figure 1) is a finite colimit for a diagram of shape ***J*** = *empty*.

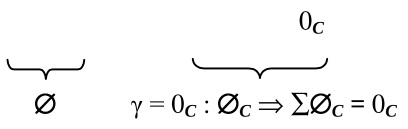

0_C
$\varnothing$
$\gamma = 0_C : \varnothing_C \Rightarrow \Sigma\varnothing_C = 0_C$

Figure 1: Initial Object

- The *empty* diagram in ***C*** is the appropriate base diagram for an initial object.

```
(1) (KIF$function diagram)
    (KIF$function empty)
    (= empty diagram)
    (= (KIF$source diagram) CAT$category)
    (= (KIF$target diagram) FUNC$functor)
    (forall (?c (CAT$category ?c))
        (= (diagram ?c)
           (the (?d ((COL$diagram ?c) ?d))
                (= ((COL$shape ?c) ?d) CAT$empty))))
```

- Initial object *cocones* are used to specify and axiomatize initial objects in a category ***C***. Each initial object cocone has the empty diagram as its base diagram and can be identified with its *opvertex* ***C***-object. An initial object cocone is a natural transformation. But since the source functor is the empty diagram, there are no component C-morphisms in this natural transformation. An initial object cocone is the very special case of a general colimit cocone over the empty diagram in ***C***.

```
(2) (KIF$function cocone)
    (= (KIF$source cocone) CAT$category)
    (= (KIF$target cocone) SET$class)
    (forall (?c (CAT$category ?c))
        (= (cocone ?c) (CAT$object ?c)))

(3) (KIF$function opvertex)
     (= (KIF$source opvertex) CAT$category)
     (= (KIF$target opvertex) SET.FTN$function)
     (forall (?c (CAT$category ?c))
         (and (= (SET.FTN$source (opvertex ?c)) (cocone ?c))
              (= (SET.FTN$target (opvertex ?c)) (CAT$object ?c))
              (= (opvertex ?c) (SET.FTN$identity (CAT$object ?c)))))
```

- There is a class of *initial* ***C***-objects (Figure 1).

```
(4) (KIF$function colimit)
    (KIF$function initial)
    (= initial colimit)
    (= (KIF$source colimit) CAT$category)
    (= (KIF$target colimit) SET$class)
    (forall (?c (CAT$category ?c))
        (SET$subclass (colimit ?c) (CAT$object ?c)))
```

- For any given object in a category ***C***, there is a function that maps any initial ***C***-object to a unique *counique* ***C***-morphism whose source is the initial object and whose target is the given object. Existence and uniqueness represents the universality of the initial object. A derived theorem states that all initial objects are isomorphic in ***C***.

```
(5) (KIF$function counique)
    (= (KIF$source counique) CAT$category)
    (= (KIF$target counique) KIF$function)
    (forall (?c (CAT$category ?c))
        (and (= (KIF$source (counique ?c)) (CAT$object ?c))
             (= (KIF$target (counique ?c)) SET.FTN$function)
             (forall (?o ((CAT$object ?c) ?o))
                 (and (= (SET.FTN$source ((counique ?c) ?o)) (colimit ?c))
                      (= (SET.FTN$target ((counique ?c) ?o)) (CAT$morphism ?c))
                      (forall (?io ((colimit ?c) ?io))
                          (= (((counique ?c) ?o) ?io)
                             (the (?m (CAT$morphism ?c) ?m))
                                  (and (= ((CAT$source ?c) ?m) ?io)
                                       (= ((CAT$target ?c) ?m) ?o)))))))))
```

Binary Coproducts

COL.COPRD

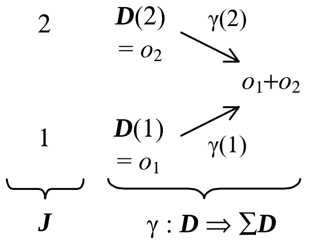


Figure 2: Binary coproduct

A *binary coproduct* in a category ***C*** (Figure 2) is a finite colimit for a diagram of shape *two* = • •. Such a diagram (of ***C***-objects and ***C***-morphisms) is called a *pair*.

- A *pair* (of ***C***-objects) is the appropriate base diagram for a binary coproduct in ***C***. Each pair consists of a pair of ***C***-objects called *object1* and *object2*. Copairs are determined by their two component ***C***-objects.

```
(1) (KIF$function diagram)
    (KIF$function pair)
    (= pair diagram)
    (= (KIF$source diagram) CAT$category)
    (= (KIF$target diagram) SET$class)

(2) (KIF$function object1)
    (= (KIF$source object1) CAT$category)
    (= (KIF$target object1) SET.FTNfunction)
    (forall (?c (CAT$category ?c))
        (and (= (SET.FTN$source (object1 ?c)) (diagram ?c))
             (= (SET.FTN$target (object1 ?c)) (CAT$object ?c))))

(3) (KIF$function object2)
    (= (KIF$source object2) CAT$category)
    (= (KIF$target object2) SET.FTNfunction)
    (forall (?c (CAT$category ?c))
        (and (= (SET.FTN$source (object2 ?c)) (diagram ?c))
             (= (SET.FTN$target (object2 ?c)) (CAT$object ?c))))

    (forall (?c (CAT$category ?c)
             ?p1 ((diagram ?c) ?p1) ?p2 ((diagram ?c) ?p2))
        (=> (and (= ((object1 ?c) ?p1) ((object1 ?c) ?p2))
                 (= ((object2 ?c) ?p1) ((object2 ?c) ?p2)))
            (= ?p1 ?p2)))
```

- Every pair has an opposite.

```
(4) (KIF$function opposite)
    (= (KIF$source opposite) CAT$category)
    (= (KIF$target opposite) SET.FTN$function)
    (forall (?c (CAT$category ?c))
        (and (= (SET.FTN$source (opposite ?c)) (diagram ?c))
             (= (SET.FTN$target (opposite ?c)) (diagram ?c))
             (= (SET.FTN$composition (opposite ?c) (object1 ?c)) (object2 ?c))
             (= (SET.FTN$composition (opposite ?c) (object2 ?c)) (object1 ?c))))
```

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

```
    (forall (?c (CAT$category ?c))
        (= (SET.FTN$composition (opposite ?c) (opposite ?c))
           (SET.FTN$identity (diagram ?c))))
```

- Binary coproduct *cocones* are used to specify and axiomatize binary coproducts in a category ***C***. Each binary coproduct cocone has an underlying binary coproduct *diagram* (*pair*) and an *opvertex* ***C***-object. A binary coproduct cocone is a natural transformation. For convenience of terminology we name the components of this natural transformation – a pair of ***C***-morphisms called *opfirst* and *opsecond*, whose common target ***C***-object is the opvertex and whose source ***C***-objects are the target ***C***-objects of the ***C***-morphisms in the underlying diagram. A binary coproduct cocone is the very special case of a general colimit cocone over a binary coproduct diagram.

```
(5) (KIF$function cocone)
    (= (KIF$source cocone) CAT$category)
    (= (KIF$target cocone) SET$class)
    (forall (?c (CAT$category ?c))
        (and (KIF$subcollection (cocone ?c) (COL$cocone ?c))
            (= (cocone ?c) ((COL$cocone-fiber ?c) CAT$two))))

(6) (KIF$function cocone-diagram)
    (= (KIF$source cocone-diagram) CAT$category)
    (= (KIF$target cocone-diagram) SET.FTN$function)
    (forall (?c (CAT$category ?c))
        (and (= (SET.FTN$source (cocone-diagram ?c)) (cocone ?c))
             (= (SET.FTN$target (cocone-diagram ?c)) (diagram ?c))
             (= (cocone-diagram ?c) ((COL$cocone-diagram-fiber ?c) CAT$two)))

(7) (KIF$function opvertex)
    (= (KIF$source opvertex) CAT$category)
    (= (KIF$target opvertex) SET.FTN$function)
    (forall (?c (CAT$category ?c))
        (and (= (SET.FTN$source (opvertex ?c)) (cocone ?c))
             (= (SET.FTN$target (opvertex ?c)) (CAT$object ?c))
             (= (opvertex ?c) ((COL$opvertex-fiber ?c) CAT$two))))

(8) (KIF$function opfirst)
    (= (KIF$source opfirst) CAT$category)
    (= (KIF$target opfirst) SET.FTN$function)
    (forall (?c (CAT$category ?c))
        (and (= (SET.FTN$source (opfirst ?c)) (cocone ?c))
             (= (SET.FTN$target (opfirst ?c)) (CAT$morphism ?c))
             (forall (?t ((cocone ?c) ?t))
                 (= ((opfirst ?c) ?t) ((NAT$component ?t) 1)))))

(9) (KIF$function opsecond)
    (= (KIF$source opsecond) CAT$category)
    (= (KIF$target opsecond) SET.FTN$function)
    (forall (?c (CAT$category ?c))
        (and (= (SET.FTN$source (opsecond ?c)) (cocone ?c))
             (= (SET.FTN$target (opsecond ?c)) (CAT$morphism ?c))
             (forall (?t ((cocone ?c) ?t))
                 (= ((opsecond ?c) ?t) ((NAT$component ?t) 2)))))
```

o There is a function '`colimiting-cocone`' that maps a binary coproduct diagram (pair) in a category ***C*** to its collection of binary coproduct colimiting cocones (this may be empty). The opvertex ***C***-object of a colimiting cocone (Figure 4) is also known a *binary coproduct*. A colimiting cocone is a natural transformation. For convenience of terminology we name the components of this natural transformation – two *injection* ***C***-morphisms.

```
(10) (KIF$function colimiting-cocone)
     (= (KIF$source colimiting-cocone) CAT$category)
     (= (KIF$target colimiting-cocone) SET.FTN$function)
     (forall (?c (CAT$category ?c))
         (and (= (SET.FTN$source (colimiting-cocone ?c)) (diagram ?c))
              (= (SET.FTN$target (colimiting-cocone ?c)) SET$class)
              (= (colimiting-cocone ?c)
                 ((COL$colimiting-cocone-fiber ?c) CAT$two))))

(11) (KIF$function colimit)
     (KIF$function binary-coproduct)
     (= binary-coproduct colimit)
     (= (KIF$source colimit) CAT$category)
     (= (KIF$target colimit) KIF$function)
     (forall (?c (CAT$category ?c))
         (and (= (KIF$source (colimit ?c)) (diagram ?c))
              (= (KIF$target (colimit ?c)) SET.FTN$function)
              (forall (?d ((diagram ?c) ?d))
                  (and (= (SET.FTN$source ((colimit ?c) ?d))
                          ((colimiting-cocone ?c) ?d))
                       (= (SET.FTN$target ((colimit ?c) ?d))
                          (CAT$object ?c))
                       (= ((colimit ?c) ?d)
                          (((COL$colimit-fiber ?c) CAT$two) ?d))))))

(12) (KIF$function injection1)
     (= (KIF$source injection1) CAT$category)
     (= (KIF$target injection1) KIF$function)
     (forall (?c (CAT$category ?c))
         (and (= (KIF$source (injection1 ?c)) (diagram ?c))
              (= (KIF$target (injection1 ?c)) SET.FTN$function)
              (forall (?d ((diagram ?c) ?d))
                  (and (= (SET.FTN$source ((injection1 ?c) ?d))
                          ((colimiting-cocone ?c) ?d))
                       (= (SET.FTN$target ((injection1 ?c) ?d))
                          (CAT$morphism ?c))
                       (forall (?g (((colimiting-cocone ?c) ?d) ?g))
                           (= (((injection1 ?c) ?d) ?g)
                              ((NAT$component ?g) 1)))))))

(13) (KIF$function injection2)
     (= (KIF$source injection2) CAT$category)
     (= (KIF$target injection2) KIF$function)
     (forall (?c (CAT$category ?c))
         (and (= (KIF$source (injection2 ?c)) (diagram ?c))
              (= (KIF$target (injection2 ?c)) SET.FTN$function)
              (forall (?d ((diagram ?c) ?d))
                  (and (= (SET.FTN$source ((injection2 ?c) ?d))
                          ((colimiting-cocone ?c) ?d))
                       (= (SET.FTN$target ((injection2 ?c) ?d))
                          (CAT$morphism ?c))
                       (forall (?g (((colimiting-cocone ?c) ?d) ?g))
                           (= (((injection2 ?c) ?d) ?g)
                              ((NAT$component ?g) 2)))))))
```

o For any binary coproduct cocone in a category ***C***, there is a function that maps any colimiting cocone with the same base diagram to a unique *comediator* ***C***-morphism whose source is the binary coproduct and whose target is opvertex of the cocone. Existence and uniqueness represents the universality of the binary coproduct operator. A derived theorem states that all binary coproducts are isomorphic in ***C***.

```
(14) (KIF$function comediator)
     (= (KIF$source comediator) CAT$category)
     (= (KIF$target comediator) KIF$function)
     (forall (?c (CAT$category ?c))
         (and (= (KIF$source (comediator ?c)) (cocone ?c))
              (= (KIF$target (comediator ?c)) SET.FTN$function)
              (forall (?t ((cocone ?c) ?t))
                  (and (= (SET.FTN$source ((comediator ?c) ?t))
                          ((colimiting-cocone ?c) ((cocone-diagram ?c) ?t)))
                       (= (SET.FTN$target ((comediator ?c) ?t)) (CAT$morphism ?c))
                       (= ((comediator ?c) ?t)
                          (((COL$comediator-fiber ?c) CAT$two) ?t))))))
```

Coequalizers

COL.COEQ

Given a category ***C***, a *coequalizer* in ***C*** (Figure 3) is a finite colimit for a diagram of shape ***J*** = $\cdot \rightrightarrows \cdot$ = *parallel-pair*. Such a diagram (of ***C***-objects and ***C***-morphisms) is also called an *parallel pair*.

o A *parallel pair* is the appropriate base diagram for a coequalizer. Each parallel pair consists of a pair of ***C***-morphisms called *first* and *second*. These are required to have a common source and a common target ***C***-object. The term ‘parallel pair’ is synonymous with the phrase “coequalizer diagram”.

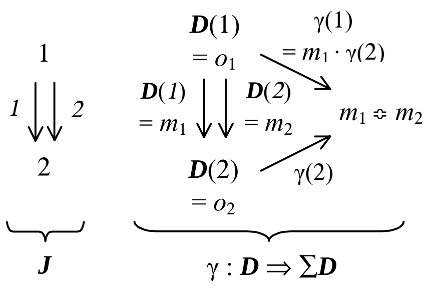


Figure 3: Coequalizer

```
(1) (KIF$function diagram)
    (KIF$function parallel-pair)
    (= parallel-pair diagram)
    (= (KIF$source diagram) CAT$category)
    (= (KIF$target diagram) SET$class)
    (forall (?c (CAT$category ?c))
        (and (KIF$subcollection (diagram ?c) (COL$diagram ?c))
             (= (diagram ?c)
                ((COL$diagram-fiber ?c) CAT$parallel-pair))))

(2) (KIF$function source)
    (= (KIF$source source) CAT$category)
    (= (KIF$target source) SET.FTN$function)
    (forall (?c (CAT$category ?c))
        (and (= (SET.FTN$source (source ?c)) (diagram ?c))
             (= (SET.FTN$target (source ?c)) (CAT$object ?c))
             (forall (?d ((diagram ?c) ?d))
                 (= ((source ?c) ?d) ((FUNC$object ?d) 1)))))

(3) (KIF$function target)
    (= (KIF$source target) CAT$category)
    (= (KIF$target target) SET.FTN$function)
    (forall (?c (CAT$category ?c))
        (and (= (SET.FTN$source (target ?c)) (diagram ?c))
             (= (SET.FTN$target (target ?c)) (CAT$object ?c))
             (forall (?d ((diagram ?c) ?d))
                 (= ((target ?c) ?d) ((FUNC$object ?d) 2)))))

(4) (KIF$function function1)
    (= (KIF$source function1) CAT$category)
    (= (KIF$target function1) SET.FTN$function)
    (forall (?c (CAT$category ?c))
        (and (= (SET.FTN$source (function1 ?c)) (diagram ?c))
             (= (SET.FTN$target (function1 ?c)) (CAT$object ?c))
             (forall (?d ((diagram ?c) ?d))
                 (= ((function1 ?c) ?d) ((FUNC$morphism ?d) 1)))))

(5) (KIF$function function2)
    (= (KIF$source function2) CAT$category)
    (= (KIF$target function2) SET.FTN$function)
    (forall (?c (CAT$category ?c))
        (and (= (SET.FTN$source (function2 ?c)) (diagram ?c))
             (= (SET.FTN$target (function2 ?c)) (CAT$object ?c))
             (forall (?d ((diagram ?c) ?d))
                 (= ((function2 ?c) ?d) ((FUNC$morphism ?d) 2)))))
```

- Coequalizer *cocones* are used to specify and axiomatize coequalizers in a category ***C***. Each coequalizer cocone has an underlying coequalizer *diagram* and an *opvertex* ***C***-object. A coequalizer cocone is a natural transformation. For convenience of terminology we name the second component of this natural transformation – a ***C***-morphism called *morphism*, whose target ***C***-object is the opvertex and whose source ***C***-object is the target of the ***C***-morphisms in the parallel-pair. By naturality, the first component is not needed – it is the composition of either parallel pair morphism with the first component. A coequalizer cocone is the very special case of a general colimit cocone over a coequalizer diagram.

```
    (6) (KIF$function cocone)
        (= (KIF$source cocone) CAT$category)
        (= (KIF$target cocone) SET$class)
        (forall (?c (CAT$category ?c))
            (and (KIF$subcollection (cocone ?c) (COL$cocone ?c))
                 (= (cocone ?c) ((COL$cocone-fiber ?c) CAT$parallel-pair))))

    (7) (KIF$function cocone-diagram)
        (= (KIF$source cocone-diagram) CAT$category)
        (= (KIF$target cocone-diagram) SET.FTN$function)
        (forall (?c (CAT$category ?c))
            (and (= (SET.FTN$source (cocone-diagram ?c)) (cocone ?c))
                 (= (SET.FTN$target (cocone-diagram ?c)) (diagram ?c))
                 (= (cocone-diagram ?c)
                    ((COL$cocone-diagram-fiber ?c) CAT$parallel-pair)))

    (8) (KIF$function opvertex)
        (= (KIF$source opvertex) CAT$category)
        (= (KIF$target opvertex) SET.FTN$function)
        (forall (?c (CAT$category ?c))
            (and (= (SET.FTN$source (opvertex ?c)) (cocone ?c))
                 (= (SET.FTN$target (opvertex ?c)) (CAT$object ?c))
                 (= (opvertex ?c) ((COL$opvertex-fiber ?c) CAT$parallel-pair))))

    (9) (KIF$function morphism)
        (= (KIF$source morphism) CAT$category)
        (= (KIF$target morphism) SET.FTN$function)
        (forall (?c (CAT$category ?c))
            (and (= (SET.FTN$source (morphism ?c)) (cocone ?c))
                 (= (SET.FTN$target (morphism ?c)) (CAT$morphism ?c))
                 (forall (?t ((cocone ?c) ?t))
                     (= ((morphism ?c) ?t) ((NAT$component ?t) 2)))))
```

- There is a function ‘`colimiting-cocone`’ that maps a coequalizer diagram (parallel pair) in a category ***C*** to its collection of coequalizer colimiting cocones (this may be empty). The opvertex ***C***-object of a colimiting cocone (Figure 3) is also known a *coequalizer*. A colimiting cocone is a natural transformation. For convenience of terminology we name the second component of this natural transformation – the *canon* ***C***-morphism.

```
(10) (KIF$function colimiting-cocone)
     (= (KIF$source colimiting-cocone) CAT$category)
     (= (KIF$target colimiting-cocone) SET.FTN$function)
     (forall (?c (CAT$category ?c))
         (and (= (SET.FTN$source (colimiting-cocone ?c)) (diagram ?c))
              (= (SET.FTN$target (colimiting-cocone ?c)) SET$class)
              (= (colimiting-cocone ?c)
                 ((COL$colimiting-cocone-fiber ?c) CAT$parallel-pair))))

(11) (KIF$function colimit)
     (KIF$function coequalizer)
     (= coequalizer colimit)
     (= (KIF$source colimit) CAT$category)
     (= (KIF$target colimit) KIF$function)
     (forall (?c (CAT$category ?c))
         (and (= (KIF$source (colimit ?c)) (diagram ?c))
              (= (KIF$target (colimit ?c)) SET.FTN$function)
              (forall (?d ((diagram ?c) ?d))
                  (and (= (SET.FTN$source ((colimit ?c) ?d))
                          ((colimiting-cocone ?c) ?d))
                       (= (SET.FTN$target ((colimit ?c) ?d))
                          (CAT$object ?c))
                       (= ((colimit ?c) ?d)
                          (((COL$colimit-fiber ?c) CAT$parallel-pair) ?d))))))

(12) (KIF$function canon)
     (= (KIF$source canon) CAT$category)
     (= (KIF$target canon) KIF$function)
     (forall (?c (CAT$category ?c))
         (and (= (KIF$source (canon ?c)) (diagram ?c))
              (= (KIF$target (canon ?c)) SET.FTN$function)
              (forall (?d ((diagram ?c) ?d))
                  (and (= (SET.FTN$source ((canon ?c) ?d))
                          ((colimiting-cocone ?c) ?d))
                       (= (SET.FTN$target ((canon ?c) ?d))
                          (CAT$morphism ?c))
                       (forall (?g (((colimiting-cocone ?c) ?d) ?g))
                           (= (((canon ?c) ?d) ?g)
                              ((NAT$component ?g) 2)))))))
```

- For any coequalizer cocone in a category ***C***, there is a function that maps any colimiting cocone with the same base diagram to a unique *comediator* ***C***-morphism whose source is the coequalizer and whose target is opvertex of the cocone. This is the unique morphism that commutes with canon and morphism. Existence and uniqueness represents the universality of the coequalizer operator. A derived theorem states that all coequalizers are isomorphic in ***C***.

```
(13) (KIF$function comediator)
     (= (KIF$source comediator) CAT$category)
     (= (KIF$target comediator) KIF$function)
     (forall (?c (CAT$category ?c))
         (and (= (KIF$source (comediator ?c)) (cocone ?c))
              (= (KIF$target (comediator ?c)) SET.FTN$function)
              (forall (?t ((cocone ?c) ?t))
                  (and (= (SET.FTN$source ((comediator ?c) ?t))
                          ((colimiting-cocone ?c) ((cocone-diagram ?c) ?t)))
                       (= (SET.FTN$target ((comediator ?c) ?t)) (CAT$morphism ?c))
                       (= ((comediator ?c) ?t)
                          (((COL$comediator-fiber ?c) CAT$parallel-pair) ?t))))))
```

Pushouts

COL.PSH

Given a category ***C***, a *pushout* in ***C*** (Figure 4) is a finite colimit for a diagram of shape ***J*** = • ← • → • = *span*. Such a diagram (of ***C***-objects and ***C***-morphisms) is also called an *span*.

$$\begin{tikzcd} 2 \\ 3 \arrow[u, "2"] \arrow[d, "1"'] \\ 1 \end{tikzcd}
\qquad
\begin{tikzcd}
 & \boldsymbol{D}(2) = o_2 \arrow[dr, "\gamma(2)"] & \\
\boldsymbol{D}(3) = o_3 \arrow[ur, "\boldsymbol{D}(a_2) = m_2"] \arrow[dr, "\boldsymbol{D}(a_1) = m_1"'] & & m_1 \diamond m_2 \\
 & \boldsymbol{D}(1) = o_1 \arrow[ur, "\gamma(1)"'] &
\end{tikzcd}$$

$$\underbrace{\qquad}_{\boldsymbol{J}} \qquad \underbrace{\qquad\qquad\qquad\qquad}_{\gamma : \boldsymbol{D} \Rightarrow \Sigma\boldsymbol{D}}$$

Figure 4: Pushout

- A *span* is the appropriate base diagram for a pushout. Each span consists of a pair of ***C***-morphisms called *first* and *second*. These are required to have a common source ***C***-object called the *vertex*. The term 'span' is synonymous with the phrase "pushout diagram".

```
(1) (KIF$function diagram)
    (KIF$function span)
    (= span diagram)
    (= (KIF$source diagram) CAT$category)
    (= (KIF$target diagram) SET$class)
    (forall (?c (CAT$category ?c))
        (and (KIF$subcollection (diagram ?c) (COL$diagram ?c))
             (= (diagram ?c)
                ((COL$diagram-fiber ?c) CAT$span))))

(2) (KIF$function object1)
    (= (KIF$source object1) CAT$category)
    (= (KIF$target object1) SET.FTN$function)
    (forall (?c (CAT$category ?c))
        (and (= (SET.FTN$source (object1 ?c)) (diagram ?c))
             (= (SET.FTN$target (object1 ?c)) (CAT$object ?c))
             (forall (?d ((diagram ?c) ?d))
                 (= ((object1 ?c) ?d) ((FUNC$object ?d) 1)))))

(3) (KIF$function object2)
    (= (KIF$source object2) CAT$category)
    (= (KIF$target object2) SET.FTN$function)
    (forall (?c (CAT$category ?c))
        (and (= (SET.FTN$source (object2 ?c)) (diagram ?c))
             (= (SET.FTN$target (object2 ?c)) (CAT$object ?c))
             (forall (?d ((diagram ?c) ?d))
                 (= ((object2 ?c) ?d) ((FUNC$object ?d) 2)))))

(4) (KIF$function vertex)
    (= (KIF$source vertex) CAT$category)
    (= (KIF$target vertex) SET.FTN$function)
    (forall (?c (CAT$category ?c))
        (and (= (SET.FTN$source (vertex ?c)) (diagram ?c))
             (= (SET.FTN$target (vertex ?c)) (CAT$object ?c))
             (forall (?d ((diagram ?c) ?d))
                 (= ((vertex ?c) ?d) ((FUNC$object ?d) 3)))))

(5) (KIF$function first)
    (= (KIF$source first) CAT$category)
    (= (KIF$target first) SET.FTN$function)
    (forall (?c (CAT$category ?c))
        (and (= (SET.FTN$source (first ?c)) (diagram ?c))
             (= (SET.FTN$target (first ?c)) (CAT$object ?c))
             (forall (?d ((diagram ?c) ?d))
                 (= ((first ?c) ?d) ((FUNC$morphism ?d) 1)))))
```

```
(6) (KIF$function second)
    (= (KIF$source second) CAT$category)
    (= (KIF$target second) SET.FTN$function)
    (forall (?c (CAT$category ?c))
        (and (= (SET.FTN$source (second ?c)) (diagram ?c))
             (= (SET.FTN$target (second ?c)) (CAT$object ?c))
             (forall (?d ((diagram ?c) ?d))
                 (= ((second ?c) ?d) ((FUNC$morphism ?d) 2)))))
```

- The *pair* of target ***C***-objects (postfixing discrete diagram) of any span (pushout diagram) in a category ***C*** is named.

```
(7) (KIF$function pair)
    (= (KIF$source pair) CAT$category)
    (= (KIF$target pair) SET.FTN$function)
    (forall (?c (CAT$category ?c))
        (and (= (SET.FTN$source (pair ?c)) (diagram ?c))
             (= (SET.FTN$target (pair ?c)) (COL.COPRD2$diagram))))
             (= (SET.FTN$composition [(pair ?c) (COL.COPRD2$object1 ?c)])
                (object1 ?c))
             (= (SET.FTN$composition [(pair ?c) (COL.COPRD2$object2 ?c)])
                (object2 ?c))))
```

- Every span in ***C*** has an opposite.

```
(8) (KIF$function opposite)
    (= (KIF$source opposite) CAT$category)
    (= (KIF$target opposite) SET.FTN$function)
    (forall (?c (CAT$category ?c))
        (and (= (SET.FTN$source (opposite ?c)) (diagram ?c))
             (= (SET.FTN$target (opposite ?c)) (diagram ?c))
             (= (SET.FTN$composition [(opposite ?c) (object1 ?c)])
                (object2 ?c))
             (= (SET.FTN$composition [(opposite ?c) (object2 ?c)])
                (object1 ?c))
             (= (SET.FTN$composition [(opposite ?c) (vertex ?c)])
                (vertex ?c))
             (= (SET.FTN$composition [(opposite ?c) (first ?c)])
                (second ?c))
             (= (SET.FTN$composition [(opposite ?c) (second ?c)])
                (first ?c))))
```

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

```
    (forall (?c (CAT$category ?c))
        (= (SET.FTN$composition [(opposite ?c) (opposite ?c)])
           (SET.FTN$identity (diagram ?c))))
```

- The *parallel pair* or *coequalizer diagram* function maps a span in a category ***C*** to the associated (`COL.COEQ`) parallel pair (see Diagram 4) of functions, which are the composite of the first and second functions of the span with the coproduct injections of the binary coproduct of the pair of classes underlying the span. The coequalizer and canon of the parallel pair could be used to define the pushout and pushout injections.

$$\begin{tikzcd}
& b \arrow[dl, "m_1"'] \arrow[d, shift left] \arrow[d, shift right] \arrow[dr, "m_2"] & \\
o_1 \arrow[r, "\iota_1"'] & o_1+o_2 & o_2 \arrow[l, "\iota_2"]
\end{tikzcd}$$

Diagram 4: Coequalizer Diagram

```
(9) (KIF$function coequalizer-diagram)
    (KIF$function parallel-pair)
    (= parallel-pair coequalizer-diagram)
    (= (KIF$source coequalizer-diagram) CAT$category)
    (= (KIF$target coequalizer-diagram) SET.FTN$function)
    (forall (?c (CAT$category ?c))
        (and (= (SET.FTN$source (coequalizer-diagram ?c)) (diagram ?c))
             (= (SET.FTN$target (coequalizer-diagram ?c)) (COL.COEQ$diagram ?c))
             (forall (?d ((diagram ?c) ?d))
                 (= ((coequalizer-diagram ?c) ?d)
                    (FUNC$composition [(FUNC$inclusion [CAT$two CAT$span]) ?d])))))
```

- Pushout *cocones* are used to specify and axiomatize pushouts in a category ***C***. Each pushout cocone has an underlying pushout *diagram* and an *opvertex* ***C***-object. A pushout cocone is a natural transformation. For convenience of terminology we name the components of this natural transformation – a pair of ***C***-morphisms called *opfirst* and *opsecond*, whose common target ***C***-object is the opvertex and whose source ***C***-objects are the target ***C***-objects of the ***C***-morphisms in the underlying diagram. By naturality, the opfirst and opsecond morphisms form a commutative diagram with the span. A pushout cocone is the very special case of a general colimit cocone over a pushout diagram.

```
(10) (KIF$function cocone)
     (= (KIF$source cocone) CAT$category)
     (= (KIF$target cocone) SET$class)
     (forall (?c (CAT$category ?c))
         (and (KIF$subcollection (cocone ?c) (COL$cocone ?c))
              (= (cocone ?c) ((COL$cocone-fiber ?c) CAT$span))))

(11) (KIF$function cocone-diagram)
     (= (KIF$source cocone-diagram) CAT$category)
     (= (KIF$target cocone-diagram) SET.FTN$function)
     (forall (?c (CAT$category ?c))
         (and (= (SET.FTN$source (cocone-diagram ?c)) (cocone ?c))
              (= (SET.FTN$target (cocone-diagram ?c)) (diagram ?c))
              (= (cocone-diagram ?c) ((COL$cocone-diagram-fiber ?c) CAT$span)))

(12) (KIF$function opvertex)
     (= (KIF$source opvertex) CAT$category)
     (= (KIF$target opvertex) SET.FTN$function)
     (forall (?c (CAT$category ?c))
         (and (= (SET.FTN$source (opvertex ?c)) (cocone ?c))
              (= (SET.FTN$target (opvertex ?c)) (CAT$object ?c))
              (= (opvertex ?c) ((COL$opvertex-fiber ?c) CAT$span))))

(13) (KIF$function opfirst)
     (= (KIF$source opfirst) CAT$category)
     (= (KIF$target opfirst) SET.FTN$function)
     (forall (?c (CAT$category ?c))
         (and (= (SET.FTN$source (opfirst ?c)) (cocone ?c))
              (= (SET.FTN$target (opfirst ?c)) (CAT$morphism ?c))
              (forall (?t ((cocone ?c) ?t))
                  (= ((opfirst ?c) ?t) ((NAT$component ?t) 1)))))

(14) (KIF$function opsecond)
     (= (KIF$source opsecond) CAT$category)
     (= (KIF$target opsecond) SET.FTN$function)
     (forall (?c (CAT$category ?c))
         (and (= (SET.FTN$source (opsecond ?c)) (cocone ?c))
              (= (SET.FTN$target (opsecond ?c)) (CAT$morphism ?c))
              (forall (?t ((cocone ?c) ?t))
                  (= ((opsecond ?c) ?t) ((NAT$component ?t) 2)))))
```

- There is a function '`colimiting-cocone`' that maps a pushout diagram (span) in a category ***C*** to its collection of pushout colimiting cocones (this may be empty). The opvertex ***C***-object of a colimiting cocone (Figure 4) is also known a *pushout*. A colimiting cocone is a natural transformation. For convenience of terminology we name the components of this natural transformation – two *injection* ***C***-morphisms.

```
(15) (KIF$function colimiting-cocone)
     (= (KIF$source colimiting-cocone) CAT$category)
     (= (KIF$target colimiting-cocone) SET.FTN$function)
     (forall (?c (CAT$category ?c))
         (and (= (SET.FTN$source (colimiting-cocone ?c)) (diagram ?c))
              (= (SET.FTN$target (colimiting-cocone ?c)) SET$class)
              (= (colimiting-cocone ?c)
                 ((COL$colimiting-cocone-fiber ?c) CAT$span))))

(16) (KIF$function colimit)
     (KIF$function pushout)
     (= pushout colimit)
     (= (KIF$source colimit) CAT$category)
     (= (KIF$target colimit) KIF$function)
     (forall (?c (CAT$category ?c))
         (and (= (KIF$source (colimit ?c)) (diagram ?c))
              (= (KIF$target (colimit ?c)) SET.FTN$function)
              (forall (?d ((diagram ?c) ?d))
                  (and (= (SET.FTN$source ((colimit ?c) ?d))
                          ((colimiting-cocone ?c) ?d))
                       (= (SET.FTN$target ((colimit ?c) ?d))
                          (CAT$object ?c))
                       (= ((colimit ?c) ?d)
                          (((COL$colimit-fiber ?c) CAT$span) ?d))))))

(17) (KIF$function injection1)
     (= (KIF$source injection1) CAT$category)
     (= (KIF$target injection1) KIF$function)
     (forall (?c (CAT$category ?c))
         (and (= (KIF$source (injection1 ?c)) (diagram ?c))
              (= (KIF$target (injection1 ?c)) SET.FTN$function)
              (forall (?d ((diagram ?c) ?d))
                  (and (= (SET.FTN$source ((injection1 ?c) ?d))
                          ((colimiting-cocone ?c) ?d))
                       (= (SET.FTN$target ((injection1 ?c) ?d))
                          (CAT$morphism ?c))
                       (forall (?g (((colimiting-cocone ?c) ?d) ?g))
                           (= (((injection1 ?c) ?d) ?g)
                              ((NAT$component ?g) 1)))))))

(18) (KIF$function injection2)
     (= (KIF$source injection2) CAT$category)
     (= (KIF$target injection2) KIF$function)
     (forall (?c (CAT$category ?c))
         (and (= (KIF$source (injection2 ?c)) (diagram ?c))
              (= (KIF$target (injection2 ?c)) SET.FTN$function)
              (forall (?d ((diagram ?c) ?d))
                  (and (= (SET.FTN$source ((injection2 ?c) ?d))
                          ((colimiting-cocone ?c) ?d))
                       (= (SET.FTN$target ((injection2 ?c) ?d))
                          (CAT$morphism ?c))
                       (forall (?g (((colimiting-cocone ?c) ?d) ?g))
                           (= (((injection2 ?c) ?d) ?g)
                              ((NAT$component ?g) 2)))))))
```

- For any pushout cocone in a category ***C***, there is a function that maps any colimiting cocone with the same base diagram to a unique *comediator* ***C***-morphism whose source is the pushout and whose target is opvertex of the cocone. This is the unique morphism that commutes with opfirst and opsecond morphisms. Existence and uniqueness represents the universality of the pullback operator. A derived theorem states that all pushouts are isomorphic in ***C***.

```
(19) (KIF$function comediator)
     (= (KIF$source comediator) CAT$category)
     (= (KIF$target comediator) KIF$function)
     (forall (?c (CAT$category ?c))
         (and (= (KIF$source (comediator ?c)) (cocone ?c))
              (= (KIF$target (comediator ?c)) SET.FTN$function)
              (forall (?t ((cocone ?c) ?t))
                  (and (= (SET.FTN$source ((comediator ?c) ?t))
                          ((colimiting-cocone ?c) ((cocone-diagram ?c) ?t)))
                       (= (SET.FTN$target ((comediator ?c) ?t)) (CAT$morphism ?c))
                       (= ((comediator ?c) ?t)
                          (((COL$comediator-fiber ?c) CAT$span) ?t))))))
```