

# The Kind Namespace

Robert E. Kent

May 9, 2006

## Contents

<b>1</b>	<b>The Kind Namespace</b>	<b>1</b>
1.1	The Kind Kernel . . . . .	2
1.1.1	Kind Sets . . . . .	3
1.1.2	Kind Predicates . . . . .	11
1.1.3	Kind Functions . . . . .	17
1.1.4	Kind Multivalued Functions . . . . .	28
1.1.5	Kind Relations . . . . .	32
1.1.6	Kind Endorelations . . . . .	45
1.2	Kind Diagrams . . . . .	48
1.2.1	Kind Set Pairs . . . . .	48
1.2.2	Kind Set Triples . . . . .	52
1.2.3	Kind Opspans . . . . .	56
1.3	Kind Limits . . . . .	61
1.3.1	Kind Binary Products . . . . .	63
1.3.2	Kind Binary Powers . . . . .	66
1.3.3	Kind Ternary Products . . . . .	68
1.3.4	Kind Ternary Powers . . . . .	72
1.3.5	Kind Pullbacks . . . . .	75
<b>A</b>	<b>Appendix</b>	<b>79</b>
A.1	Unions and Universes . . . . .	79

## 1 The Kind Namespace

### Namespace Prefix

**Technical:** kind  
**Recommended:** kind

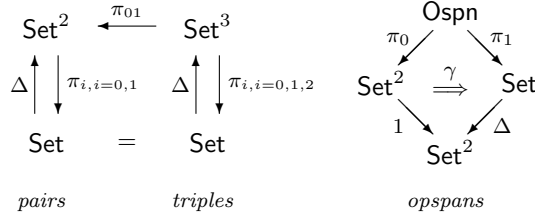


Figure 5: Categories of Diagrams

## 1.2 Kind Diagrams

### Namespace Prefix

**Technical:** kind.dgm

**Recommended:** kind.dgm

The nested namespace for finite kind diagrams essentially defines three categories of diagrams (Figure 5): the category  $\text{Set}^2$  of set pairs and function pairs (diagrams for binary products), the category  $\text{Set}^3$  of set triples and function triples (diagrams for ternary products), and the (comma) category  $\text{Ospn} = (1, \Delta)$  of opspans and opspan morphisms (diagrams for pullbacks). The terminology of this namespace, which is listed in Table 3, consists of 65 terms and 59 concepts (6 synonyms).

There are three basic kinds of things and their morphisms: a collection of kind *set pairs* with kind *function pairs* as their morphisms, a collection of kind *set triples* with kind *function triples* as their morphisms, and a collection of kind *opspans* with kind *opspan morphisms* as their morphisms. All six are distinct – these sets are pairwise disjoint. All three basic kinds have suitable category-theoretic maps, such as *source*, *target*, *composition* and *identity*.

Some of the components of these collections are also introduced here. There are *set0* and *set1* maps from set pairs to sets, and *function0* and *function1* maps from function pairs to functions. There are *set0*, *set1* and *set2* maps from set triples to sets, and *function0*, *function1* and *function2* maps from function triples to functions. There are *opzeroth* and *opfirst* maps from opspans to functions, *opvertex*, *set0* and *set1* maps from opspans to sets, and *function0* and *function1* maps from opspan morphisms to functions.

### 1.2.1 Kind Set Pairs

**Objects.** A kind *set pair*  $(X_0, X_1)$  is a pair of kind sets  $X_0, X_1 \in \text{set}$ . The collection of set pairs  $\text{set}^2 = \text{set} \times \text{set} = \{(X_0, X_1) \mid X_0, X_1 \in \text{set}\}$  is the binary power of  $\text{set}$ . The set pairs that are axiomatized here are concrete: they can be referenced as ‘[?X0 ?X1]’.

```
(iff:set set-pair)
(forall (?X (set-pair ?X))
  (exists (?X0 (kind.set:set ?X0) ?X1 (kind.set:set ?X1))
```

(Emphasized terms are IFF terms.)

	IFF Set	IFF Function	IFF Relation
pr.obj	set-pair	set0 set1 constant	
pr.mor	function-pair	function0 function1 source target constant	
	composable-pair	factor0 factor1 composition identity	composable
trp.obj	set-triple	set0 set1 set2 = set set-pair constant	
trp.mor	function-triple	function0 function1 function2 = function function-pair source target constant	
	composable-pair	factor0 factor1 composition identity	composable
ospn.obj	opspan	set-pair set0 set1 set = opvertex constant function-pair function0 = opzeroth function1 = opfirst	
ospn.mor	opspan-morphism	function-pair function0 function1 function = opvertex source target constant	
	composable-pair	factor0 factor1 composition identity	composable

Technical and Recommended Prefix : kind.dgm

Table 3: The Finite Diagram Kind Namespace

```
(= ?X [?X0 ?X1]))
(forall (?X0 (kind.set:set ?X0) ?X1 (kind.set:set ?X1))
  (set-pair [?X0 ?X1]))
```

Each set pair consists of a pair of sets called *set0* and *set1*.

```
(iff:function set0)
(= (iff:source set0) set-pair)
(= (iff:target set0) kind.set:set)
(forall (?X0 (kind.set:set ?X0) ?X1 (kind.set:set ?X1))
  (= (set0 [?X0 ?X1]) ?X0))
```

```
(iff:function set1)
(= (iff:source set1) set-pair)
(= (iff:target set1) kind.set:set)
(forall (?X0 (kind.set:set ?X0) ?X1 (kind.set:set ?X1))
  (= (set1 [?X0 ?X1]) ?X1))
```

The *constant* function  $\Delta : \text{set} \rightarrow \text{set}^2$  maps a set  $X$  to the set pair  $(X, X)$ . This is the object function of a functor  $\Delta : \text{Set} \rightarrow \text{Set}^2$ .

```
(iff:function constant)
(= (iff:source constant) kind.set:set)
(= (iff:target constant) set-pair)
(forall (?X (kind.set:set ?X))
  (and (= (set0 (constant ?X)) ?X)
    (= (set1 (constant ?X)) ?X)))
```

**Morphisms.** A kind *function pair*  $(f_0, f_1)$  is a pair of kind functions  $f_0, f_1 \in \text{ftn}$ . The collection  $\text{ftn}^2 = \text{ftn} \times \text{ftn} = \{(f_0, f_1) \mid f_0, f_1 \in \text{ftn}\}$  is the binary power of  $\text{ftn}$ . The function pairs that are axiomatized here are concrete: they can be referenced as '[?f0 ?f1]'.

```
(iff:set function-pair)
(forall (?f (function-pair ?f))
```

```

    (exists (?f0 (kind.ftn:function ?f0) ?f1 (kind.set:function ?f1))
      (= ?f [?f0 ?f1]))
(forall (?f0 (kind.ftn:function ?f0) ?f1 (kind.set:function ?f1))
  (function-pair [?f0 ?f1]))

```

Each function pair consists of a pair of functions called *function0* and *function1*.

```

(iff:function function0)
(= (iff:source function0) function-pair)
(= (iff:target function0) kind.ftn:function)
(forall (?f0 (kind.ftn:function ?f0) ?f1 (kind.ftn:function ?f1))
  (= (function0 [?f0 ?f1]) ?f0))

```

```

(iff:function function1)
(= (iff:source function1) function-pair)
(= (iff:target function1) kind.ftn:function)
(forall (?f0 (kind.ftn:function ?f0) ?f1 (kind.ftn:function ?f1))
  (= (function1 [?f0 ?f1]) ?f1))

```

Each function pair  $(f_0 : X_0 \rightarrow Y_0, f_1 : X_1 \rightarrow Y_1)$  has an underlying source set pair  $(X_0, X_1)$  and target set pair  $(Y_0, Y_1)$ .

```

(iff:function source)
(= (iff:source source) function-pair)
(= (iff:target source) kind.dgm.pr.obj:set-pair)
(forall (?f (function-pair ?f))
  (and (= (kind.dgm.pr.obj:set0 (source ?f)) (kind.ftn:source (function0 ?f)))
    (= (kind.dgm.pr.obj:set1 (source ?f)) (kind.ftn:source (function1 ?f)))))

```

```

(iff:function target)
(= (iff:source target) function-pair)
(= (iff:target target) kind.dgm.pr.obj:set-pair)
(forall (?f (function-pair ?f))
  (and (= (kind.dgm.pr.obj:set0 (target ?f)) (kind.ftn:target (function0 ?f)))
    (= (kind.dgm.pr.obj:set1 (target ?f)) (kind.ftn:target (function1 ?f)))))

```

The *constant* function  $\Delta : \text{ftn} \rightarrow \text{ftn}^2$  maps a function  $f$  to the function pair  $(f, f)$ . This is the morphism function of a functor  $\Delta : \text{Set} \rightarrow \text{Set}^2$ .

```

(iff:function constant)
(= (iff:source constant) kind.ftn:function)
(= (iff:target constant) function-pair)
(forall (?f (kind.ftn:function ?f))
  (and (= (function0 (constant ?f)) ?f)
    (= (function1 (constant ?f)) ?f)))

```

**Category Theory.** Two kind function pairs are *composable* when the target of the first is equal to the source of the second.

```

(iff:relation composable)
(= (iff:set0 composable) function-pair)
(= (iff:set1 composable) function-pair)
(forall (?f (function-pair ?f) ?g (function-pair ?g))
  (<=> (composable ?f ?g)
    (= (target ?f) (source ?g))))

```

We name the extent and projection factors of the composable endorelation.

```

(iff:set composable-pair)
(forall (?fg (composable-pair ?fg))
  (exists (?f (function-pair ?f)) ?g (function-pair ?g))
    (= ?fg [?f ?g]))
(forall (?f (function-pair ?f)) ?g (function-pair ?g))
  (<=> (composable-pair [?f ?g])
    (composable ?f ?g))

(iff:function factor0)
(= (iff:source factor0) composable-pair)
(= (iff:target factor0) function-pair)
(forall (?f (function-pair ?f)) ?g (function-pair ?g) (composable-pair [?f ?g]))
  (= (factor0 [?f ?g]) ?f))

(iff:function factor1)
(= (iff:source factor1) composable-pair)
(= (iff:target factor1) function-pair)
(forall (?f (function-pair ?f)) ?g (function-pair ?g) (composable-pair [?f ?g]))
  (= (factor1 [?f ?g]) ?g))

```

The *composition* of two composable kind function pairs is defined factorwise. The source of the composite is the source of the first factor, and the target of the composite is the target of the second factor. Composition is surjective (see identity properties below). Composition is associative.

```

(iff:function composition)
(= (iff:source composition) composable-pair)
(= (iff:target composition) function-pair)
(forall (?f (function-pair ?f)) ?g (function-pair ?g) (composable ?f ?g))
  (and (= (source (composition [?f ?g])) (source ?f))
    (= (target (composition [?f ?g])) (target ?g))
    (= (function0 (composition [?f ?g]))
      (kind.ftn:composition [(function0 ?f) (function0 ?g)]))
    (= (function1 (composition [?f ?g]))
      (kind.ftn:composition [(function1 ?f) (function1 ?g)])))
(forall (?h (function-pair ?h))
  (exists (?f (function-pair ?f)) ?g (function-pair ?g) (composable ?f ?g))
    (= (composition [?f ?g]) ?h)))
(forall (?f (function-pair ?f)) ?g (function-pair ?g) ?h (function-pair ?h))
  (composable ?f ?g) (composable ?g ?h))
(= (composition [?f (composition [?g ?h])])
  (composition [(composition [?f ?g]) ?h]))

```

For every kind set pair, there is a unique associated *identity* kind function pair. The identity on any set pair is defined factorwise. Composition with the identity of the source (target) of a function pair returns that function pair. Identity is injective; hence, set pairs can be regarded as special function pairs that satisfy the unit laws.

```

(iff:function identity)
(= (iff:source identity) kind.dgm.pr.obj:set-pair)
(= (iff:target identity) function-pair)
(forall (?X (kind.dgm.pr.obj:set-pair ?X))
  (and (= (source (identity ?X)) ?X)
    (= (target (identity ?X)) ?X)
    (= (function0 (identity ?X))
      (kind.ftn:identity (kind.dgm.pr.obj:set0 ?X)))
    (= (function1 (identity ?X))

```

```

      (kind.ftn:identity (kind.dgm.pr.obj:set1 ?X))))))
(forall (?X (kind.dgm.pr.obj:set-pair ?X))
  (kind.ftn:bijective (identity ?X)))
(forall (?X0 (kind.dgm.pr.obj:set-pair ?X0) ?X1 (kind.dgm.pr.obj:set-pair ?X1))
  (= (identity ?X0) (identity ?X1)))
(= ?X0 ?X1))
(forall (?f (function-pair ?f))
  (and (= (composition [(identity (source ?f)) ?f]) ?f)
    (= ?f (composition [?f (identity (target ?f))])))))

```

## 1.2.2 Kind Set Triples

**Objects.** A kind *set triple*  $(X_0, X_1, X_2)$  is a triple of kind sets  $X_0, X_1, X_2 \in \text{set}$ . The collection  $\text{set}^3 = \text{set} \times \text{set} \times \text{set} = \{(X_0, X_1, X_2) \mid X_0, X_1, X_2 \in \text{set}\}$  is the ternary power of  $\text{set}$ . The set triples that are axiomatized here are concrete: they can be referenced as ‘[?X0 ?X1 ?X2]’.

```

(iff:set set-triple)
(forall (?X (set-triple ?X))
  (exists (?X0 (kind.set:set ?X0) ?X1 (kind.set:set ?X1) ?X2 (kind.set:set ?X2))
    (= ?X [?X0 ?X1 ?X2])))
(forall (?X0 (kind.set:set ?X0) ?X1 (kind.set:set ?X1) ?X2 (kind.set:set ?X2))
  (set-triple [?X0 ?X1 ?X2]))

(iff:function set0)
(= (iff:source set0) set-triple)
(= (iff:target set0) kind.set:set)
(forall (?X0 (kind.set:set ?X0) ?X1 (kind.set:set ?X1) ?X2 (kind.set:set ?X2))
  (= (set0 [?X0 ?X1 ?X2]) ?X0))

(iff:function set1)
(= (iff:source set1) set-triple)
(= (iff:target set1) kind.set:set)
(forall (?X0 (kind.set:set ?X0) ?X1 (kind.set:set ?X1) ?X2 (kind.set:set ?X2))
  (= (set1 [?X0 ?X1 ?X2]) ?X1))

(iff:function set2)
(= (iff:source set2) set-triple)
(= (iff:target set2) kind.set:set)
(forall (?X0 (kind.set:set ?X0) ?X1 (kind.set:set ?X1) ?X2 (kind.set:set ?X2))
  (= (set2 [?X0 ?X1 ?X2]) ?X2))

```

Set triples can be partitioned in several ways into set pairs and single sets. We choose one way, since all the rest are equivalent. Each set triple  $(X_0, X_1, X_2)$  can be partitioned into a set pair  $(X_0, X_1)$  and a single set  $X_2$ . This corresponds to the isomorphism  $\text{set}^3 \cong \text{set}^2 \times \text{set}$ .

```

(iff:function set-pair)
(= (iff:source set-pair) set-triple)
(= (iff:target set-pair) kind.dgm.pr.obj:set-pair)
(forall (?X (set-triple ?X))
  (and (= (kind.dgm.pr.obj:set0 (set-pair ?X)) (set0 ?X))
    (= (kind.dgm.pr.obj:set1 (set-pair ?X)) (set1 ?X))))

(iff:function set)
(= (iff:source set) set-triple)

```

```
(= (iff:target set) kind.set:set)
(= set set2)
```

The *constant* function  $\Delta : \text{set} \rightarrow \text{set}^3$  maps a set  $X$  to the set triple  $(X, X, X)$ . This is the object function of a functor  $\Delta : \text{Set} \rightarrow \text{Set}^3$ .

```
(iff:function constant)
(= (iff:source constant) kind.set:set)
(= (iff:target constant) set-triple)
(forall (?X (kind.set:set ?X))
  (and (= (set0 (constant ?X)) ?X)
        (= (set1 (constant ?X)) ?X)
        (= (set2 (constant ?X)) ?X)))
```

**Morphisms.** A kind *function triple*  $(f_0, f_1, f_2)$  is a triple of kind functions  $f_0, f_1, f_2 \in \text{ftn}$ . The collection  $\text{ftn}^3 = \text{ftn} \times \text{ftn} \times \text{ftn} = \{(f_0, f_1, f_2) \mid f_0, f_1, f_2 \in \text{ftn}\}$  is the ternary power of  $\text{ftn}$ . The function triples that are axiomatized here are concrete: they can be referenced as  $[\text{?f0 ?f1 ?f2}]$ .

```
(iff:set function-triple)
(forall (?f (function-triple ?f))
  (exists (?f0 (kind.ftn:function ?f0) ?f1 (kind.set:function ?f1) ?f2 (kind.set:function ?f2))
    (= ?f [?f0 ?f1 ?f2])))
(forall (?f0 (kind.ftn:function ?f0) ?f1 (kind.set:function ?f1) ?f2 (kind.set:function ?f2))
  (function-triple [?f0 ?f1 ?f2]))
```

```
(iff:function function0)
(= (iff:source function0) function-triple)
(= (iff:target function0) kind.ftn:function)
(forall (?f0 (kind.ftn:function ?f0) ?f1 (kind.ftn:function ?f1) ?f2 (kind.ftn:function ?f2))
  (= (function0 [?f0 ?f1 ?f2]) ?f0))
```

```
(iff:function function1)
(= (iff:source function1) function-triple)
(= (iff:target function1) kind.ftn:function)
(forall (?f0 (kind.ftn:function ?f0) ?f1 (kind.ftn:function ?f1) ?f2 (kind.ftn:function ?f2))
  (= (function1 [?f0 ?f1 ?f2]) ?f1))
```

```
(iff:function function2)
(= (iff:source function2) function-triple)
(= (iff:target function2) kind.ftn:function)
(forall (?f0 (kind.ftn:function ?f0) ?f1 (kind.ftn:function ?f1) ?f2 (kind.ftn:function ?f2))
  (= (function2 [?f0 ?f1 ?f2]) ?f2))
```

Each function triple  $(f_0 : X_0 \rightarrow Y_0, f_1 : X_1 \rightarrow Y_1, f_2 : X_2 \rightarrow Y_2)$  has a source set triple  $(X_0, X_1, X_2)$  and target set triple  $(Y_0, Y_1, Y_2)$ .

```
(iff:function source)
(= (iff:source source) function-triple)
(= (iff:target source) kind.dgm.trp.obj:set-triple)
(forall (?f (function-triple ?f))
  (and (= (kind.dgm.trp.obj:set0 (source ?f)) (kind.ftn:source (function0 ?f)))
        (= (kind.dgm.trp.obj:set1 (source ?f)) (kind.ftn:source (function1 ?f)))
        (= (kind.dgm.trp.obj:set2 (source ?f)) (kind.ftn:source (function2 ?f)))))
```

```
(iff:function target)
(= (iff:source target) function-triple)
```

```

(= (iff:target target) kind.dgm.trp.obj:set-triple)
(forall (?f (function-triple ?f))
  (and (= (kind.dgm.trp.obj:set0 (target ?f)) (kind.ftn:target (function0 ?f)))
        (= (kind.dgm.trp.obj:set1 (target ?f)) (kind.ftn:target (function1 ?f)))
        (= (kind.dgm.trp.obj:set2 (target ?f)) (kind.ftn:target (function2 ?f)))))

```

The *constant* function  $\Delta : \text{ftn} \rightarrow \text{ftn}^3$  maps a function  $f$  to the function triple  $(f, f, f)$ . This is the morphism function of a functor  $\Delta : \text{Set} \rightarrow \text{Set}^3$ .

```

(iff:function constant)
(= (iff:source constant) kind.ftn:function)
(= (iff:target constant) function-triple)
(forall (?f (kind.ftn:function ?f))
  (and (= (function0 (constant ?f)) ?f)
        (= (function1 (constant ?f)) ?f)
        (= (function2 (constant ?f)) ?f)))

```

Function triples can be partitioned in several ways into function pairs and single functions. We choose one way, since all the rest are equivalent. Each function triple  $(f_0, f_1, f_2)$  can be partitioned into a function pair  $(f_0, f_1)$  and a single function  $f_2$ . This corresponds to the isomorphism  $\text{ftn}^3 \cong \text{ftn}^2 \times \text{ftn}$ .

```

(iff:function function-pair)
(= (iff:source function-pair) function-triple)
(= (iff:target function-pair) kind.dgm.pr.mor:function-pair)
(forall (?f (function-triple ?f))
  (and (= (kind.dgm.pr.mor:source (function-pair ?f)) (kind.dgm.pr.obj:set-pair (source ?f)))
        (= (kind.dgm.pr.mor:target (function-pair ?f)) (kind.dgm.pr.obj:set-pair (target ?f)))
        (= (kind.dgm.pr.mor:function0 (function-pair ?f)) (function0 ?f))
        (= (kind.dgm.pr.mor:function1 (function-pair ?f)) (function1 ?f))))

(iff:function function)
(= (iff:source function) function-triple)
(= (iff:target function) kind.ftn:function)
(= function function2)

```

**Category Theory.** Two kind function triples are *composable* when the target of the first is equal to the source of the second.

```

(iff:relation composable)
(= (iff:set0 composable) function-triple)
(= (iff:set1 composable) function-triple)
(forall (?f (function-triple ?f) ?g (function-triple ?g))
  (<=> (composable ?f ?g)
        (= (target ?f) (source ?g))))

```

We name the extent and projection factors of the composable endorelation.

```

(iff:set composable-pair)
(forall (?fg (composable-pair ?fg))
  (exists (?f (function-triple ?f)) ?g (function-triple ?g)
    (= ?fg [?f ?g])))
(forall (?f (function-triple ?f)) ?g (function-triple ?g)
  (<=> (composable-pair [?f ?g])
        (composable ?f ?g)))

(iff:function factor0)
(= (iff:source factor0) composable-pair)

```

```

(= (iff:target factor0) function-triple)
(forall (?f (function-triple ?f)) ?g (function-triple ?g) (composable-pair [?f ?g]))
  (= (factor0 [?f ?g]) ?f))

(iff:function factor1)
(= (iff:source factor1) composable-pair)
(= (iff:target factor1) function-triple)
(forall (?f (function-triple ?f)) ?g (function-triple ?g) (composable-pair [?f ?g]))
  (= (factor1 [?f ?g]) ?g))

```

The *composition* of two composable kind function triples is defined factorwise. The source of the composite is the source of the first factor, and the target of the composite is the target of the second factor. Composition is surjective (see identity properties below). Composition is associative.

```

(iff:function composition)
(= (iff:source composition) composable-pair)
(= (iff:target composition) function-triple)
(forall (?f (function-triple ?f) ?g (function-triple ?g) (composable ?f ?g))
  (and (= (source (composition [?f ?g])) (source ?f))
    (= (target (composition [?f ?g])) (target ?g))
    (= (function-pair (composition [?f ?g]))
      (kind.dgm.pr.mor:composition [(function-pair ?f) (function-pair ?g)]))
    (= (function (composition [?f ?g]))
      (kind.ftn:composition [(function ?f) (function ?g)]))))))
(forall (?h (function-triple ?h))
  (exists (?f (function-triple ?f) ?g (function-triple ?g) (composable ?f ?g))
    (= (composition [?f ?g]) ?h)))
(forall (?f (function-triple ?f) ?g (function-triple ?g) ?h (function-triple ?h)
  (composable ?f ?g) (composable ?g ?h))
  (= (composition [?f (composition [?g ?h])])
    (composition [(composition [?f ?g]) ?h])))

```

For every kind set triple, there is a unique associated *identity* kind function triple. The identity on any set triple is defined factorwise. Composition with the identity of the source (target) of a function triple returns that function triple. Identity is injective; hence, set triples can be regarded as special function triples that satisfy the unit laws.

```

(iff:function identity)
(= (iff:source identity) kind.dgm.trp.obj:set-triple)
(= (iff:target identity) function-triple)
(forall (?X (kind.dgm.trp.obj:set-triple ?X))
  (and (= (source (identity ?X)) ?X)
    (= (target (identity ?X)) ?X)
    (= (function-pair (identity ?X))
      (kind.dgm.pr.mor:identity (kind.dgm.trp.obj:set-pair ?X)))
    (= (function (identity ?X))
      (kind.ftn:identity (kind.dgm.trp.obj:set ?X)))))
(forall (?X (kind.dgm.trp.obj:set-triple ?X))
  (kind.ftn:bijective (identity ?X)))
(forall (?X0 (kind.dgm.trp.obj:set-triple ?X0) ?X1 (kind.dgm.trp.obj:set-triple ?X1))
  (= (identity ?X0) (identity ?X1)))
  (= ?X0 ?X1))
(forall (?f (function-triple ?f))
  (and (= (composition [(identity (source ?f)) ?f]) ?f)
    (= ?f (composition [?f (identity (target ?f))])))

```

### 1.2.3 Kind Opspans

**Objects.** An *opspan* is an object in the comma category

$$\text{Set}^2 \xleftarrow{\pi_0} (1, \Delta) \xrightarrow{\pi_1} \text{Set}$$

over the functorial ospan  $1_{\text{Set}^2} : \text{Set}^2 \rightarrow \text{Set}^2 \leftarrow \text{Set} : \Delta$ . More specifically, an opspan is a triple  $((X_0, X_1), X, (x_0, x_1))$  consisting of a set pair  $(X_0, X_1)$ , an opvertex set  $X$  and a function pair  $(x_0, x_1) : (X_0, X_1) \rightarrow \Delta(X)$ . The first definition below expresses these observations; however, in order to make the definition concrete, we use the second definition below, which is isomorphic.

$$\begin{aligned} \widehat{\text{opspn}} &= \{((X_0, X_1), X, (x_0, x_1)) \mid \\ &\quad X_0, X_1, X \in \text{set}, x_0, x_1 \in \text{ftn}, \\ &\quad \partial_0(x_0) = X_0, \partial_1(x_0) = X, \partial_0(x_1) = X_1, \partial_1(x_1) = X\}, \text{ or} \\ \text{ospn} &= \{(x_0, x_1) \mid x_0, x_1 \in \text{ftn}, \partial_1(x_0) = \partial_1(x_1)\} \subseteq \text{ftn} \times \text{ftn}. \end{aligned}$$

The map  $\widehat{\text{opspn}} \rightarrow \text{ospn}$  is just projection; the map  $\text{ospn} \rightarrow \widehat{\text{opspn}}$  is defined by  $(x_0, x_1) \mapsto ((\partial_0(x_0), \partial_0(x_1)), \partial_1(x_0) = \partial_1(x_1), (x_0, x_1))$ . Opspans are used as the diagrams for pullbacks. We name the projections. The opspans that are axiomatized here are concrete: they can be referenced in a form such as

```
(kind.ftn:function ?x0)
(kind.ftn:function ?x1)
(= (kind.ftn:target ?x0) (kind.ftn:target ?x1))
(kind.set:set ?Y)
(= ?Y (kind.lim.pbk.obj:pullback [?x0 ?x1]))
```

Here is the axiomatization.

```
(iff:set opspan)
(iff:subset opspan kind.dgm.pr.mor:function-pair)
(forall (?x (kind.dgm.pr.mor:function-pair ?x))
  (<=> (opspan ?x)
    (= (kind.ftn:target (kind.dgm.pr.mor:function0 ?x))
      (kind.ftn:target (kind.dgm.pr.mor:function1 ?x)))))

(iff:function set-pair)
(= (iff:source set-pair) opspan)
(= (iff:target set-pair) kind.dgm.pr.obj:set-pair)
(forall (?x (opspan ?x))
  (= (set-pair ?x) (kind.dgm.pr.mor:source ?x)))

(iff:function set0)
(= (iff:source set0) opspan)
(= (iff:target set0) kind.set:set)
(forall (?x (opspan ?x))
  (= (set0 ?x) (kind.dgm.pr.obj:set0 (set-pair ?x))))

(iff:function set1)
(= (iff:source set1) opspan)
(= (iff:target set1) kind.set:set)
(forall (?x (opspan ?x))
  (= (set1 ?x) (kind.dgm.pr.obj:set1 (set-pair ?x))))

(iff:function set) (iff:function opvertex) (= opvertex set)
```

```

(= (iff:source set) opspan)
(= (iff:target set) kind.set:set)
(forall (?x (opspan ?x))
  (= (kind.dgm.pr.obj:constant (set ?x)) (kind.dgm.pr.mor:target ?x)))

(iff:function function-pair)
(= (iff:source function-pair) opspan)
(= (iff:target function-pair) kind.dgm.pr.mor:function-pair)
(forall (?x (opspan ?x))
  (= (function-pair ?x) ?x))

(iff:function function0) (iff:function opzeroth) (= opzeroth function0)
(= (iff:source function0) opspan)
(= (iff:target function0) kind.ftn:function)
(forall (?x (opspan ?x))
  (and (= (kind.ftn:source (function0 ?x)) (set0 ?x))
        (= (kind.ftn:target (function0 ?x)) (set ?x))
        (= (function0 ?x) (kind.dgm.pr.mor:function0 ?x))))

(iff:function function1) (iff:function opfirst) (= opfirst function1)
(= (iff:source function1) opspan)
(= (iff:target function1) kind.ftn:function)
(forall (?x (opspan ?x))
  (and (= (kind.ftn:source (function1 ?x)) (set1 ?x))
        (= (kind.ftn:target (function1 ?x)) (set ?x))
        (= (function1 ?x) (kind.dgm.pr.mor:function1 ?x))))

```

The *constant* function  $\Delta : \text{set} \rightarrow \text{ospn}$  maps a set  $X$  to the opspan  $((X, X), X, (1_x, 1_x))$ . This is the object function of a functor  $\Delta : \text{Set} \rightarrow \text{Ospn}$ .

```

(iff:function constant)
(= (iff:source constant) kind.set:set)
(= (iff:target constant) opspan)
(forall (?X (kind.set:set ?X))
  (and (= (set0 (constant ?X)) ?X)
        (= (set1 (constant ?X)) ?X)
        (= (set (constant ?X)) ?X)
        (= (function0 (constant ?X)) (kind.ftn:identity ?X))
        (= (function1 (constant ?X)) (kind.ftn:identity ?X))))

```

**Morphisms.** A kind *opspan morphism* is a morphism in the comma category

$$\text{Set}^2 \xleftarrow{\pi_0} (1, \Delta) \xrightarrow{\pi_1} \text{Set}$$

over the functorial opspan  $1_{\text{Set}^2} : \text{Set}^2 \rightarrow \text{Set}^2 \leftarrow \text{Set} : \Delta$ . More specifically, an opspan morphism from source opspan  $(x_0, x_1) : (X_0, X_1) \rightarrow \Delta(X)$  to target opspan  $(y_0, y_1) : (Y_0, Y_1) \rightarrow \Delta(Y)$  is a pair  $((f_0, f_1), f)$  consisting of a function pair  $(f_0, f_1) : (X_0, X_1) \rightarrow (Y_0, Y_1)$  and an (opvertex) function  $f : X \rightarrow Y$ , which satisfy the following commutativity condition in the category  $\text{Set}^2$

$$\begin{array}{ccc}
(X_0, X_1) & \xrightarrow{(x_0, x_1)} & \Delta(X) \\
(f_0, f_1) \downarrow & & \downarrow \Delta(f) \\
(Y_0, Y_1) & \xrightarrow{(y_0, y_1)} & \Delta(Y)
\end{array}$$

Let  $\text{ospn-mor} \subseteq \text{ospn} \times \text{ftn}^2 \times \text{ftn} \times \text{ospn}$  denote the collection of opspan morphisms. We name the projections

$$\begin{aligned}\pi_{01} &: \text{ospn-mor} \rightarrow \text{ftn}^2 \\ \pi &: \text{ospn-mor} \rightarrow \text{ftn} \\ \partial_0 &: \text{ospn-mor} \rightarrow \text{ospn} \\ \partial_1 &: \text{ospn-mor} \rightarrow \text{ospn}.\end{aligned}$$

That is, each opspan morphism  $((x_0, x_1), (f_0, f_1), f, (y_0, y_1))$  has a function pair  $(f_0, f_1)$ , a function  $f$ , a source opspan  $(x_0, x_1)$  and a target opspan  $(y_0, y_1)$ .

```
(iff:set opspan-morphism)
(forall (?om (opspan-morphism ?om))
  (and (= (kind.dgm.pr.mor:source (function-pair ?om)) (kind.dgm.ospn.obj:set-pair (source ?om)))
        (= (kind.dgm.pr.mor:target (function-pair ?om)) (kind.dgm.ospn.obj:set-pair (target ?om)))
        (= (kind.ftn:source (function ?om)) (kind.dgm.ospn.obj:set (source ?om)))
        (= (kind.ftn:target (function ?om)) (kind.dgm.ospn.obj:set (target ?om)))
        (= (kind.dgm.pr.mor:composition
            [(function-pair ?om) (kind.dgm.ospn.obj:function-pair (target ?om))])
            (kind.dgm.pr.mor:composition
              [(kind.dgm.ospn.obj:function-pair (source ?om))
               (kind.dgm.pr.mor:constant (function ?om))])))
  (forall (?x01 (kind.dgm.ospn.obj:opspan ?x01)
           ?y01 (kind.dgm.ospn.obj:opspan ?y01)
           ?f01 (kind.dgm.pr.mor:function-pair ?f01)
           ?f (kind.ftn:function ?f))
    (= (kind.dgm.pr.mor:source ?f01) (kind.dgm.ospn.obj:set-pair ?x01))
    (= (kind.dgm.pr.mor:target ?f01) (kind.dgm.ospn.obj:set-pair ?y01))
    (= (kind.ftn:source ?f) (kind.dgm.ospn.obj:set ?x01))
    (= (kind.ftn:target ?f) (kind.dgm.ospn.obj:set ?y01))
    (= (kind.dgm.pr.mor:composition
        [?f01 (kind.dgm.ospn.obj:function-pair ?y01)])
        (kind.dgm.pr.mor:composition
          [(kind.dgm.ospn.obj:function-pair ?x01)
           (kind.dgm.pr.mor:constant ?f)])))
    (and (opspan-morphism [?x01 ?f01 ?f ?y01])
          (= (source [?x01 ?f01 ?f ?y01]) ?x01)
          (= (function-pair [?x01 ?f01 ?f ?y01]) ?f01)
          (= (function [?x01 ?f01 ?f ?y01]) ?f)
          (= (target [?x01 ?f01 ?f ?y01]) ?y01)))

(iff:function function-pair)
(= (iff:source function-pair) opspan-morphism)
(= (iff:target function-pair) kind.dgm.pr.mor:function-pair)

(iff:function function0)
(= (iff:source function0) opspan-morphism)
(= (iff:target function0) kind.ftn:function)
(forall (?om (opspan-morphism ?om))
  (= (function0 ?om) (kind.dgm.pr.mor:function0 (function-pair ?om))))

(iff:function function1)
(= (iff:source function1) opspan-morphism)
(= (iff:target function1) kind.ftn:function)
(forall (?om (opspan-morphism ?om))
  (= (function1 ?om) (kind.dgm.pr.mor:function1 (function-pair ?om))))

(iff:function function) (iff:function opvertex) (= opvertex function)
```

```

(= (iff:source function) opspan-morphism)
(= (iff:target function) kind.ftn:function)

(iff:function source)
(= (iff:source source) opspan-morphism)
(= (iff:target source) kind.dgm.ospn.obj:opspan)

(iff:function target)
(= (iff:source target) opspan-morphism)
(= (iff:target target) kind.dgm.ospn.obj:opspan)

```

The *constant* function  $\Delta : \text{ftn} \rightarrow \text{ospn-mor}$  maps a function  $f$  to the opspan morphism  $((f, f), f)$ . This is the morphism function of a functor  $\Delta : \text{Set} \rightarrow \text{Ospn}$ .

```

(iff:function constant)
(= (iff:source constant) kind.ftn:function)
(= (iff:target constant) opspan-morphism)
(forall (?f (kind.ftn:function ?f))
  (and (= (function0 (constant ?f)) ?f)
        (= (function1 (constant ?f)) ?f)
        (= (function (constant ?f)) ?f)))

```

**Category Theory.** Two kind opspan morphisms are *composable* when the target of the first is equal to the source of the second.

```

(iff:relation composable)
(= (iff:set0 composable) opspan-morphism)
(= (iff:set1 composable) opspan-morphism)
(forall (?f (opspan-morphism ?f) ?g (opspan-morphism ?g))
  (<=> (composable ?f ?g)
        (= (target ?f) (source ?g))))

```

We name the extent and projection factors of the composable endorelation.

```

(iff:set composable-pair)
(forall (?fg (composable-pair ?fg))
  (exists (?f (opspan-morphism ?f)) ?g (opspan-morphism ?g)
    (= ?fg [?f ?g])))
(forall (?f (opspan-morphism ?f)) ?g (opspan-morphism ?g)
  (<=> (composable-pair [?f ?g])
        (composable ?f ?g)))

(iff:function factor0)
(= (iff:source factor0) composable-pair)
(= (iff:target factor0) opspan-morphism)
(forall (?f (opspan-morphism ?f)) ?g (opspan-morphism ?g) (composable-pair [?f ?g]))
  (= (factor0 [?f ?g]) ?f))

(iff:function factor1)
(= (iff:source factor1) composable-pair)
(= (iff:target factor1) opspan-morphism)
(forall (?f (opspan-morphism ?f)) ?g (opspan-morphism ?g) (composable-pair [?f ?g]))
  (= (factor1 [?f ?g]) ?g))

```

The *composition* of two composable kind opspan morphisms is defined factorwise. The source of the composite is the source of the first factor, and the target of the composite is the target of the second factor. Composition is surjective (see identity properties below). Composition is associative.

```

(iff:function composition)
(= (iff:source composition) composable-pair)
(= (iff:target composition) opspan-morphism)
(forall (?f (opspan-morphism ?f) ?g (opspan-morphism ?g) (composable ?f ?g))
  (and (= (source (composition [?f ?g])) (source ?f))
    (= (target (composition [?f ?g])) (target ?g))
    (= (function-pair (composition [?f ?g]))
      (kind.dgm.pr.mor:composition [(function-pair ?f) (function-pair ?g)]))
    (= (function (composition [?f ?g]))
      (kind.ftn:composition [(function ?f) (function ?g)]))))
(forall (?h (opspan-morphism ?h))
  (exists (?f (opspan-morphism ?f) ?g (opspan-morphism ?g) (composable ?f ?g))
    (= (composition [?f ?g] ?h))))
(forall (?f (opspan-morphism ?f) ?g (opspan-morphism ?g) ?h (opspan-morphism ?h)
  (composable ?f ?g) (composable ?g ?h))
  (= (composition [?f (composition [?g ?h])])
    (composition [(composition [?f ?g] ?h)])))

```

For every kind opspan, there is a unique associated *identity* kind opspan morphism. The identity on any opspan is defined factorwise. Composition with the identity of the source (target) of a opspan morphism returns that opspan morphism. Identity is injective; hence, opsans can be regarded as special opspan morphisms that satisfy the unit laws.

```

(iff:function identity)
(= (iff:source identity) kind.dgm.ospn.obj:opspan)
(= (iff:target identity) opspan-morphism)
(forall (?o (kind.dgm.ospn.obj:opspan ?o))
  (and (= (source (identity ?o)) ?o)
    (= (target (identity ?o)) ?o)
    (= (function-pair (identity ?o))
      (kind.dgm.pr.mor:identity (kind.dgm.ospn.obj:set-pair ?o)))
    (= (function (identity ?o))
      (kind.ftn:identity (kind.dgm.ospn.obj:set ?o)))))
(forall (?o (kind.dgm.ospn.obj:opspan ?o))
  (kind.ftn:bijective (identity ?o)))
(forall (?o0 (kind.dgm.ospn.obj:opspan ?o0) ?o1 (kind.dgm.ospn.obj:opspan ?o1))
  (= (identity ?o0) (identity ?o1)))
(= ?o0 ?o1))
(forall (?om (opspan-morphism ?om))
  (and (= (composition [(identity (source ?om)) ?om] ?om)
    (= ?om (composition [?om (identity (target ?om))])))))

```

Our two standard references for the IFF are the books: *Sets for Mathematics* (2003) by F. William Lawvere and Robert Rosebrugh [1] and *Categories for the Working Mathematician* (1971) by Saunders Mac Lane [2].

## References

- [1] F. William Lawvere and Robert Rosebrugh. *Sets for Mathematics*. Cambridge University Press, 2003.
- [2] Saunders Mac Lane. *Categories for the Working Mathematician*. Springer-Verlag, 1971.