

The IFF Type Namespace

Robert E. Kent

December 18, 2007

Contents

1	The Core Component	2
1.1	Introduction	2
1.2	Type Sets	9
1.3	Type Functions	21
1.3.1	Function Morphisms	35
1.4	Type Spans	40
1.4.1	Type Endospans	55
1.4.2	Span Morphisms	57
1.5	Type Predicates	65
1.6	Type Relations	73
1.6.1	Type Endorelations	85
1.7	Type Diagrams	88
1.7.1	Category Theory	88
1.7.2	Introduction	88
1.7.3	Terminology	90
1.7.4	Type Set Pairs	91
1.7.5	Type Set Triples	95
1.7.6	Type Parallel Pairs	99
1.7.7	Type Opspans	105
1.8	Type Limits	112
1.8.1	Introduction	112
1.8.2	Type Binary Products	115
1.8.3	Type Binary Powers	123
1.8.4	Type Ternary Products	127
1.8.5	Type Ternary Powers	134
1.8.6	Type Equalizers	138
1.8.7	Type Pullbacks	146

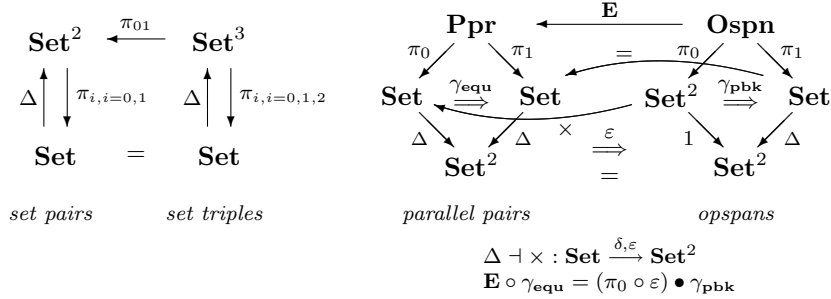


Figure 13: Categories of Diagrams

1.7 Type Diagrams

Namespace Prefix

Technical: type.dgm

Recommended: type.dgm

1.7.1 Category Theory

The nested namespace for finite type diagrams essentially defines four categories of diagrams (Figure 13): the category \mathbf{Set}^2 of set pairs and function pairs (diagrams for binary products), the category \mathbf{Set}^3 of set triples and function triples (diagrams for ternary products), the (comma) category $\mathbf{Ppr} = (\Delta, \Delta)$ of parallel pairs and parallel pair morphisms (diagrams for equalizers), and the (comma) category $\mathbf{Ospn} = (1, \Delta)$ of opspans and opspan morphisms (diagrams for pullbacks).

1.7.2 Introduction

Abstractly, a finite type diagram is a graph morphism from a finite (shape) graph into the underlying graph $\mathbf{U}(\mathbf{Set})$ of type sets and their functions. Diagrams are determined by their shape, set and function components. Here we only introduce those diagrams used in lower metalevels for axiomatizing specific finite limits. These diagrams include the empty diagram, set pairs and triples, parallel pairs of functions and opspans; their limits are called the terminal set, binary products, ternary products, equalizers and pullbacks, respectively. In Figure 14, the finite diagrams consist of a collection of sets represent by small disks and a collection of connecting edges, the limits are represented by small filled squares with projections from limit to node sets in the underlying diagram, and cones consist of a vertex set represented by a small circle with component functions from vertex set to node sets in the underlying diagram. All subdiagrams in Figure 14 are commutative.

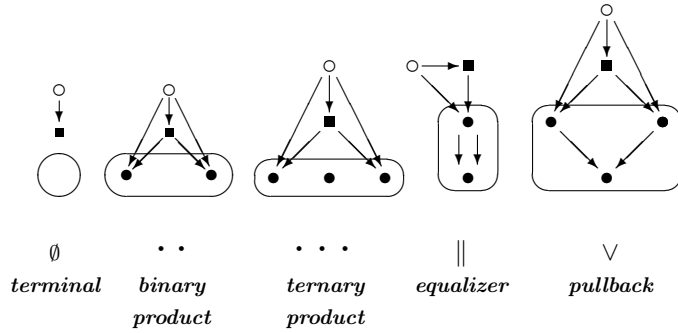


Figure 14: Finite Diagrams, Limits, Projections, Cones and Mediators

(Emphasized terms are IFF terms.)

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

Technical and Recommended Prefix : type.dgm

Table 11: The Finite Diagram Type Namespace

1.7.3 Terminology

The terminology of this namespace, which is listed in Table 11, consists of 91 terms and 85 concepts (6 synonyms).

There are four basic types of things and their morphisms: a collection of type *set pairs* with type *function pairs* as their morphisms, a collection of type *set triples* with type *function triples* as their morphisms, a collection of type *parallel pairs* with type *parallel pair morphisms* as their morphisms, and a collection of type *ospans* with type *opspan morphisms* as their morphisms. All eight are distinct — these sets are pairwise disjoint. All four basic types have the category-theoretic maps of *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 *function0* and *function1* maps from parallel pairs to functions, *set0* and *set1* maps from parallel pairs to sets, and *function0* and *function1* maps from parallel pair morphisms to functions. There are *opzeroth* and *opfirst* maps from ospans to functions, *opvertex*, *set0* and *set1* maps from ospans to sets, and *opvertex*, *function0* and *function1* maps from opspan morphisms to functions.

There are also a few conversion functions from one basic type to another: set pairs and ospans can be flipped over (they have *opposites*), any set has four *constant* basic types, any set pair determines an *opspan* with terminal *opvertex*, and (by taking the product of its associated set pair) any opspan has an associated *parallel pair*. Conversion between basic types provides a connection between their limits.

1.7.4 Type Set Pairs

`type.dgm.pr`

Set pairs are used as the diagrams for binary products.

Objects.

`type.dgm.pr.obj`

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

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

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

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

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

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

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

A pair of sets has an associated *opspan*. The *opvertex* is the terminal set, and the component functions are the unique functions for the component sets in the pair. There is an *opspan* function $\vee : \mathbf{set}^2 \rightarrow \mathbf{ospn}$, which is the object function of the *opspan* functor $\vee : \mathbf{Set}^2 \rightarrow \mathbf{Ospn}$.

```
(iff:function opspan)
(= (iff:source opspan) set-pair)
(= (iff:target opspan) type.dgm.ospn.obj:opspan)
(forall ((set-pair ?X))
  (and (= (type.dgm.ospn.obj:set0 (opspan ?X)) (set0 ?X))
        (= (type.dgm.ospn.obj:set1 (opspan ?X)) (set1 ?X))
        (= (type.dgm.ospn.obj:opvertex (opspan ?X)) type.set:terminal)
        (= (type.dgm.ospn.obj:function0 (opspan ?X)) (type.set:unique (set0 ?X)))
        (= (type.dgm.ospn.obj:function1 (opspan ?X)) (type.set:unique (set1 ?X))))))
```

For any set pair $X = (X_0, X_1)$, there is an *opposite* set pair $X^{\text{op}} = (X_1, X_0)$. This defines the opposite function $\alpha : \mathbf{set}^2 \rightarrow \mathbf{set}^2$, which is the object function of the involution functor $\alpha : \mathbf{Set}^{2\text{op}} \rightarrow \mathbf{Set}^2$.

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

The opposite of the opposite is the original set pair.

```
(forall ((set-pair ?X))
  (= (opposite (opposite ?X)) ?X))
```

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

```
(iff:set function-pair)
(forall ((function-pair ?f))
  (exists ((type.ftn:function ?f0) (type.set:function ?f1))
    (= ?f [?f0 ?f1])))
(forall ((type.ftn:function ?f0) (type.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) type.ftn:function)
(forall ((type.ftn:function ?f0) (type.ftn:function ?f1))
  (= (function0 [?f0 ?f1]) ?f0))

(iff:function function1)
(= (iff:source function1) function-pair)
(= (iff:target function1) type.ftn:function)
(forall ((type.ftn:function ?f0) (type.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) type.dgm.pr.obj:set-pair)
(forall ((function-pair ?f))
  (and (= (type.dgm.pr.obj:set0 (source ?f)) (type.ftn:source (function0 ?f)))
        (= (type.dgm.pr.obj:set1 (source ?f)) (type.ftn:source (function1 ?f)))))

(iff:function target)
(= (iff:source target) function-pair)
(= (iff:target target) type.dgm.pr.obj:set-pair)
(forall ((type.ftn:function ?f0) (type.ftn:function ?f1))
  (and (= (type.dgm.pr.obj:set0 (target ?f)) (type.ftn:target (function0 ?f)))
        (= (type.dgm.pr.obj:set1 (target ?f)) (type.ftn:target (function1 ?f)))))
```

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

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

A pair of functions has an associated *opspan morphism*. The opvertex is the terminal set identity, and the component functions are the pair. This defines the opspan morphism function $\vee : \mathbf{ftn}^2 \rightarrow \mathbf{ospn-mor}$, which is the morphism function of the opspan functor $\vee : \mathbf{Set}^2 \rightarrow \mathbf{Ospn}$.

```
(iff:function opspan-morphism)
(= (iff:source opspan-morphism) function-pair)
(= (iff:target opspan-morphism) type.dgm.ospn.mor:opspan-morphism)
(forall ((type.ftn:function-pair ?f))
  (and (= (type.dgm.ospn.mor:source (opspan-morphism ?f))
          (type.dgm.ospn.obj:opspan (source ?f)))
        (= (type.dgm.ospn.mor:target (opspan-morphism ?f))
          (type.dgm.ospn.mor:opspan (target ?f)))
        (= (type.dgm.ospn.mor:function-pair (opspan-morphism ?f)) ?f)
        (= (type.dgm.ospn.mor:opvertex (opspan-morphism ?f))
          (type.ftn:identity type.set:terminal))))
```

For any function pair $f = (f_0, f_1)$, there is an *opposite* function pair $f^{\text{op}} = (f_1, f_0)$. This defines the opposite function $\propto : \mathbf{ftn}^2 \rightarrow \mathbf{ftn}^2$, which is the morphism function of the involution functor $\propto : \mathbf{Set}^{2\text{op}} \rightarrow \mathbf{Set}^2$.

```
(iff:function opposite)
(= (iff:source opposite) function-pair)
(= (iff:target opposite) function-pair)
(forall ((function-pair ?f))
  (and (= (function0 (opposite ?f)) (set1 ?f))
        (= (function1 (opposite ?f)) (set0 ?f))))
```

The opposite of the opposite is the original function pair.

```
(forall ((function-pair ?f))
  (= (opposite (opposite ?f)) ?f))
```

Category Theory. Two type function pairs are *composable* when the target of the first is equal to the source of the second. We name the extent and projection factors of the composable endorelation.

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

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

```

(= (iff:target factor0) function-pair)
(forall ((function-pair ?f) (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 ((function-pair ?f) (function-pair ?g) (composable-pair [?f ?g]))
  (= (factor1 [?f ?g]) ?g))

```

The *composition* of two composable type 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 ((function-pair ?f) (function-pair ?g) (composable-pair [?f ?g]))
  (and (= (source (composition [?f ?g])) (source ?f))
    (= (target (composition [?f ?g])) (target ?g))
    (= (function0 (composition [?f ?g]))
      (type.ftn:composition [(function0 ?f) (function0 ?g)]))
    (= (function1 (composition [?f ?g]))
      (type.ftn:composition [(function1 ?f) (function1 ?g)]))))
(forall ((function-pair ?h))
  (exists ((function-pair ?f) (function-pair ?g) (composable-pair [?f ?g]))
    (= (composition [?f ?g]) ?h)))
(forall ((function-pair ?f) (function-pair ?g) (function-pair ?h)
  (composable-pair [?f ?g]) (composable-pair [?g ?h]))
  (= (composition [?f (composition [?g ?h])])
    (composition [(composition [?f ?g]) ?h])))

```

For every type set pair, there is a unique associated *identity* type function pair. The identity on any set pair is defined factorwise. Composition satisfies two unit laws: 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) type.dgm.pr.obj:set-pair)
(= (iff:target identity) function-pair)
(forall ((type.dgm.pr.obj:set-pair ?X))
  (and (= (source (identity ?X)) ?X)
    (= (target (identity ?X)) ?X)
    (= (function0 (identity ?X)) (type.ftn:identity (type.dgm.pr.obj:set0 ?X)))
    (= (function1 (identity ?X)) (type.ftn:identity (type.dgm.pr.obj:set1 ?X)))))
(forall ((type.dgm.pr.obj:set-pair ?X))
  (type.ftn:bijective (identity ?X)))
(forall ((type.dgm.pr.obj:set-pair ?X0) (type.dgm.pr.obj:set-pair ?X1))
  (= (identity ?X0) (identity ?X1)))
(= ?X0 ?X1)
(forall ((function-pair ?f))
  (and (= (composition [(identity (source ?f)) ?f]) ?f)
    (= ?f (composition [?f (identity (target ?f))]))))

```

1.7.5 Type Set Triples

type.dgm.trp

Set triples are used as the diagrams for ternary products.

Objects.

type.dgm.trp.obj

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

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

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

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

(iff:function set2)
(= (iff:source set2) set-triple)
(= (iff:target set2) type.set:set)
(forall ((type.set:set ?X0) (type.set:set ?X1) (type.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 $\mathbf{set}^3 \cong \mathbf{set}^2 \times \mathbf{set}$.

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

(iff:function set)
(= (iff:source set) set-triple)
(= (iff:target set) type.set:set)
(= set set2)
```

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

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

Morphisms. A type *function triple* (f_0, f_1, f_2) is a triple of type functions $f_0, f_1, f_2 \in \mathbf{ftn}$. The collection of function triples $\mathbf{ftn}^3 = \mathbf{ftn} \times \mathbf{ftn} \times \mathbf{ftn} = \{(f_0, f_1, f_2) \mid f_0, f_1, f_2 \in \mathbf{ftn}\}$ is the ternary power of \mathbf{ftn} . The function triples that are axiomatized here are concrete: they can be referenced as '[?f0 ?f1 ?f2]'.⁷

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

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

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

```
(iff:function function2)
(= (iff:source function2) function-triple)
(= (iff:target function2) type.ftn:function)
(forall ((type.ftn:function ?f0) (type.ftn:function ?f1) (type.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) type.dgm.trp.obj:set-triple)
(forall ((function-triple ?f))
  (and (= (type.dgm.trp.obj:set0 (source ?f)) (type.ftn:source (function0 ?f)))
        (= (type.dgm.trp.obj:set1 (source ?f)) (type.ftn:source (function1 ?f)))
        (= (type.dgm.trp.obj:set2 (source ?f)) (type.ftn:source (function2 ?f)))))
```

```
(iff:function target)
(= (iff:source target) function-triple)
(= (iff:target target) type.dgm.trp.obj:set-triple)
(forall ((function-triple ?f))
  (and (= (type.dgm.trp.obj:set0 (target ?f)) (type.ftn:target (function0 ?f)))
```

```

(= (type.dgm.trp.obj:set1 (target ?f)) (type.ftn:target (function1 ?f)))
(= (type.dgm.trp.obj:set2 (target ?f)) (type.ftn:target (function2 ?f))))

```

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

```

(iff:function constant)
(= (iff:source constant) type.ftn:function)
(= (iff:target constant) function-triple)
(forall ((type.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 $\mathbf{ftn}^3 \cong \mathbf{ftn}^2 \times \mathbf{ftn}$.

```

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

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

```

Category Theory. Two type function triples are *composable* when the target of the first is equal to the source of the second. We name the extent and projection factors of the composable endorelation.

```

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

(iff:function factor0)
(= (iff:source factor0) composable-pair)
(= (iff:target factor0) function-triple)
(forall ((function-triple ?f) (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 ((function-triple ?f) (function-triple ?g) (composable-pair [?f ?g]))
  (= (factor1 [?f ?g]) ?g))

```

The *composition* of two composable type 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 ((function-triple ?f) (function-triple ?g) (composable-pair [?f ?g]))
  (and (= (source (composition [?f ?g])) (source ?f))
        (= (target (composition [?f ?g])) (target ?g))
        (= (function-pair (composition [?f ?g]))
            (type.dgm.pr.mor:composition [(function-pair ?f) (function-pair ?g)]))
        (= (function (composition [?f ?g]))
            (type.ftn:composition [(function ?f) (function ?g)]))))
(forall ((function-triple ?h)
  (exists ((function-triple ?f) (function-triple ?g) (composable-pair [?f ?g]))
    (= (composition [?f ?g] ?h))))
(forall ((function-triple ?f) (function-triple ?g) (function-triple ?h)
  (composable-pair [?f ?g]) (composable-pair [?g ?h]))
  (= (composition [?f (composition [?g ?h])]
      (composition [(composition [?f ?g] ?h)])))
```

For every type set triple, there is a unique associated *identity* type function triple. The identity on any set triple is defined factorwise. Composition satisfies two unit laws: 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) type.dgm.trp.obj:set-triple)
(= (iff:target identity) function-triple)
(forall ((type.dgm.trp.obj:set-triple ?X))
  (and (= (source (identity ?X)) ?X)
        (= (target (identity ?X)) ?X)
        (= (function-pair (identity ?X))
            (type.dgm.pr.mor:identity (type.dgm.trp.obj:set-pair ?X)))
        (= (function (identity ?X))
            (type.ftn:identity (type.dgm.trp.obj:set ?X))))
(forall ((type.dgm.trp.obj:set-triple ?X))
  (type.ftn:bijjective (identity ?X)))
(forall ((type.dgm.trp.obj:set-triple ?X0) (type.dgm.trp.obj:set-triple ?X1)
  (= (identity ?X0) (identity ?X1)))
  (= ?X0 ?X1))
(forall ((function-triple ?f))
  (and (= (composition [(identity (source ?f)) ?f] ?f) ?f)
        (= ?f (composition [?f (identity (target ?f))]))))
```

1.7.6 Type Parallel Pairs

`type.dgm.ppr`

Parallel pairs are used as the diagrams for equalizers.

Objects.

`type.dgm.ppr.obj`

A *parallel pair* is an object in the comma category

$$\mathbf{Set} \xleftarrow{\pi_0} (\Delta, \Delta) \xrightarrow{\pi_1} \mathbf{Set}$$

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

$$\begin{aligned} \widehat{\mathbf{ppr}} &= \{(X_0, X_1, (x_0, x_1)) \mid \\ &\quad X_0, X_1 \in \mathbf{set}, (x_0, x_1) \in \mathbf{ftn}^2, \\ &\quad \partial_0(x_0, x_1) = \Delta(X_0), \partial_1(x_0, x_1) = \Delta(X_1)\}, \text{ or} \\ \mathbf{ppr} &= \{(x_0, x_1) \mid x_0, x_1 \in \mathbf{ftn}, \partial_0(x_0) = \partial_0(x_1), \partial_1(x_0) = \partial_1(x_1)\} \subseteq \mathbf{ftn} \times \mathbf{ftn}. \end{aligned}$$

The map $\widehat{\mathbf{ppr}} \rightarrow \mathbf{ppr}$ is just projection; the map $\mathbf{ppr} \rightarrow \widehat{\mathbf{ppr}}$ is defined by $(x_0, x_1) \mapsto (\partial_0(x_0) = \partial_0(x_1), \partial_1(x_0) = \partial_1(x_1), (x_0, x_1))$. We name the projections. The parallel pairs that are axiomatized here are concrete: they can be referenced in a form such as

```
(type.ftn:function ?x0)
(type.ftn:function ?x1)
(= (type.ftn:source ?x0) (type.ftn:source ?x1))
(= (type.ftn:target ?x0) (type.ftn:target ?x1))
(type.set:set ?Y)
(= ?Y (type.lim.equ.obj:equalizer [?x0 ?x1]))
```

Here is the axiomatization.

```
(iff:set parallel-pair)
(forall ((parallel-pair ?x)) (type.dgm.pr.mor:function-pair ?x))
(forall ((type.ftn:function ?x0) (type.ftn:function ?x1))
  (<=> (parallel-pair [?x0 ?x1])
    (and (= (type.ftn:source ?x0) (type.ftn:source ?x1))
         (= (type.ftn:target ?x0) (type.ftn:target ?x1)))))

(iff:function function0)
(= (iff:source function0) parallel-pair)
(= (iff:target function0) type.ftn:function)
(forall ((type.ftn:function ?x0) (type.ftn:function ?x1) (parallel-pair [?x0 ?x1]))
  (= (function0 [?x0 ?x1]) ?x0))

(iff:function function1)
(= (iff:source function1) parallel-pair)
(= (iff:target function1) type.ftn:function)
```

```

(forall ((type.ftn:function ?x0) (type.ftn:function ?x1) (parallel-pair [?x0 ?x1]))
  (= (function1 [?x0 ?x1]) ?x1))

(iff:function set0)
(= (iff:source set0) parallel-pair)
(= (iff:target set0) type.set:set)
(forall ((parallel-pair ?x))
  (and (= (set0 ?x) (type.ftn:source (function0 ?x)))
        (= (set0 ?x) (type.ftn:source (function1 ?x)))))

(iff:function set1)
(= (iff:source set1) parallel-pair)
(= (iff:target set1) type.set:set)
(forall ((parallel-pair ?x))
  (and (= (set1 ?x) (type.ftn:target (function0 ?x)))
        (= (set1 ?x) (type.ftn:target (function1 ?x)))))

(iff:function set-pair)
(= (iff:source set-pair) parallel-pair)
(= (iff:target set-pair) type.dgm.pr.obj:set-pair)
(forall ((parallel-pair ?x))
  (and (= (type.dgm.pr.obj:set0 (set-pair ?x)) (set0 ?x))
        (= (type.dgm.pr.obj:set1 (set-pair ?x)) (set1 ?x))))

```

The *constant* function $\Delta : \mathbf{set} \rightarrow \mathbf{ppr}$ maps a set X to the parallel pair $(X, X, (1_X, 1_X))$. This is the object function of the constant functor $\Delta : \mathbf{Set} \rightarrow \mathbf{Ppr}$.

```

(iff:function constant)
(= (iff:source constant) type.set:set)
(= (iff:target constant) parallel-pair)
(forall ((type.set:set ?X))
  (and (= (set0 (constant ?X)) ?X)
        (= (set1 (constant ?X)) ?X)
        (= (function0 (constant ?X)) (type.ftn:identity ?X))
        (= (function1 (constant ?X)) (type.ftn:identity ?X))))

```

Morphisms.

`type.dgm.ppr.mor`

A type *parallel pair morphism* is a morphism in the comma category

$$\mathbf{Set} \xleftarrow{\pi_0} (\Delta, \Delta) \xrightarrow{\pi_1} \mathbf{Set}$$

over the functorial ospan $\Delta : \mathbf{Set} \rightarrow \mathbf{Set}^2 \leftarrow \mathbf{Set} : \Delta$. More specifically, a parallel pair morphism $\phi : X \rightarrow Y$ is a triple $\phi = (X, f, Y)$, consisting of a *source* parallel pair $X = (x_0, x_1) : \Delta(X_0) \rightarrow \Delta(X_1)$, a *target* parallel pair $Y = (y_0, y_1) : \Delta(Y_0) \rightarrow \Delta(Y_1)$ and is a *function pair* $f = (f_0, f_1)$, $f_0 : X_0 \rightarrow Y_0$ and $f_1 : X_1 \rightarrow Y_1$, which satisfy the following commutativity condition in the category \mathbf{Set}^2

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

Thus, any parallel pair morphism has an underlying function pair (f_0, f_1) that forms a commuting diagram with the component functions of the source and target, $x_0 \cdot f_1 = f_0 \cdot y_0$ and $x_1 \cdot f_1 = f_0 \cdot y_1$, and has a source (target) parallel pair, whose set components consist of the sources (targets) of its function components. Let $\mathbf{ppr}\text{-mor} \subseteq \mathbf{ppr} \times \mathbf{ftn}^2 \times \mathbf{ppr}$ denote the collection of parallel pair morphisms. We name the projections

$$\begin{aligned}\pi_0 &: \mathbf{ppr}\text{-mor} \rightarrow \mathbf{ftn} \\ \pi_1 &: \mathbf{ppr}\text{-mor} \rightarrow \mathbf{ftn} \\ \partial_0 &: \mathbf{ppr}\text{-mor} \rightarrow \mathbf{ppr} \\ \partial_1 &: \mathbf{ppr}\text{-mor} \rightarrow \mathbf{ppr}.\end{aligned}$$

That is, each parallel pair morphism $f = (X, f_{01}, Y)$ has a function-pair $f_{01} = (f_0, f_1)$, a source parallel pair $X = (x_0, x_1)$ and a target parallel pair $Y = (y_0, y_1)$. Parallel pair morphisms are used in defining the packing-unpacking isomorphism of equalizers.

```
(iff:set parallel-pair-morphism)
(forall ((parallel-pair-morphism ?f))
  (exists ((type.dgm.ppr.obj:parallel-pair ?X)
    (type.dgm.pr.mor:function-pair ?f01)
    (type.dgm.ppr.obj:parallel-pair ?Y))
    (= ?f [?X ?f01 ?Y])))
(forall ((type.dgm.ppr.obj:parallel-pair ?X)
  (type.dgm.pr.mor:function-pair ?f01)
  (type.dgm.ppr.obj:parallel-pair ?Y))
  (<=> (parallel-pair-morphism [?X ?f01 ?Y])
    (= (type.dgm.pr.mor:composition
      [(type.dgm.ppr.obj:function-pair ?X)
      (type.dgm.ppr.mor:constant (type.dgm.pr.mor:function1 ?f01))])
      (type.dgm.pr.mor:composition
      [(type.dgm.ppr.mor:constant (type.dgm.pr.mor:function0 ?f01))
      (type.dgm.ppr.obj:function-pair ?Y)]))))))

(iff:function function-pair)
(= (iff:source function-pair) parallel-pair-morphism)
(= (iff:target function-pair) type.dgm.pr.mor:function-pair)
(forall ((type.dgm.ppr.obj:parallel-pair ?X)
  (type.dgm.pr.mor:function-pair ?f01)
  (type.dgm.ppr.obj:parallel-pair ?Y)
  (parallel-pair-morphism [?X ?f01 ?Y]))
  (= (function-pair [?X ?f01 ?Y]) ?f01))

(iff:function function0)
(= (iff:source function0) parallel-pair-morphism)
(= (iff:target function0) type.ftn:function)
(forall ((parallel-pair-morphism ?f))
  (= (function0 ?f) (type.dgm.pr.mor:function0 (function-pair ?f))))

(iff:function function1)
(= (iff:source function1) parallel-pair-morphism)
(= (iff:target function1) type.ftn:function)
(forall ((parallel-pair-morphism ?f))
  (= (function1 ?f) (type.dgm.pr.mor:function1 (function-pair ?f))))

(iff:function source)
(= (iff:source source) parallel-pair-morphism)
```

```

(= (iff:target source) type.dgm.ppr.obj:parallel-pair)
(forall ((type.dgm.ppr.obj:parallel-pair ?X)
         (type.dgm.pr.mor:function-pair ?f01)
         (type.dgm.ppr.obj:parallel-pair ?Y)
         (parallel-pair-morphism [?X ?f01 ?Y]))
  (= (source [?X ?f01 ?Y]) ?X))

(iff:function target)
(= (iff:source target) parallel-pair-morphism)
(= (iff:target target) type.dgm.ppr.obj:parallel-pair)
(forall ((type.dgm.ppr.obj:parallel-pair ?X)
         (type.dgm.pr.mor:function-pair ?f01)
         (type.dgm.ppr.obj:parallel-pair ?Y)
         (parallel-pair-morphism [?X ?f01 ?Y]))
  (= (target [?X ?f01 ?Y]) ?Y))

(forall ((parallel-pair-morphism ?ppm)
         (and (= (type.ftn:source (function0 ?ppm)) (type.dgm.ppr.obj:set0 (source ?ppm)))
              (= (type.ftn:target (function0 ?ppm)) (type.dgm.ppr.obj:set0 (target ?ppm)))
              (= (type.ftn:source (function1 ?ppm)) (type.dgm.ppr.obj:set1 (source ?ppm)))
              (= (type.ftn:target (function1 ?ppm)) (type.dgm.ppr.obj:set1 (target ?ppm)))
              (= (type.ftn:composition [(type.dgm.ppr.obj:function0 (source ?ppm)) (function1 ?ppm)])
                  (type.ftn:composition [(function0 ?ppm) (type.dgm.ppr.obj:function0 (target ?ppm))]))
              (= (type.ftn:composition [(type.dgm.ppr.obj:function1 (source ?ppm)) (function1 ?ppm)])
                  (type.ftn:composition [(function0 ?ppm) (type.dgm.ppr.obj:function1 (target ?ppm))])))))

```

The *constant* function $\Delta : \mathbf{ftn} \rightarrow \mathbf{ppr}\text{-}\mathbf{mor}$ maps a function $f : X \rightarrow Y$ to the parallel pair morphism $\Delta(f) = (\Delta(X), (f, f), \Delta(Y)) : \Delta(X) \rightarrow \Delta(Y)$. This is the morphism function of the constant functor $\Delta : \mathbf{Set} \rightarrow \mathbf{Ppr}$.

```

(iff:function constant)
(= (iff:source constant) type.ftn:function)
(= (iff:target constant) parallel-pair-morphism)
(forall ((type.ftn:function ?f)
         (and (= (source (constant ?f)) (type.dgm.ppr.obj:constant (type.ftn:source ?f)))
              (= (target (constant ?f)) (type.dgm.ppr.obj:constant (type.ftn:target ?f)))
              (= (function0 (constant ?f)) ?f)
              (= (function1 (constant ?f)) ?f))))

```

Category Theory. Parallel pair morphisms can be composed. Two type parallel pair morphisms are *composable* when the target of the first is equal to the source of the second. We name the extent and projection factors of the composable endorelation.

```

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

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

```

```

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

```

The *composition* $f \cdot g : X \rightarrow Z$ of two composable type parallel pair morphisms, $f = (f_0, f_1) : X \rightarrow Y$ and $g = (g_0, g_1) : Y \rightarrow Z$, is defined factorwise: $(f \cdot g)_0 = f_0 \cdot g_0$ and $(f \cdot g)_1 = f_1 \cdot g_1$. 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) parallel-pair-morphism)
(forall ((parallel-pair-morphism ?f) (parallel-pair-morphism ?g)
         (composable-pair [?f ?g]))
  (and (= (source (composition [?f ?g])) (source ?f))
        (= (target (composition [?f ?g])) (target ?g))
        (= (function-pair (composition [?f ?g]))
            (type.dgm.pr.mor:composition [(function-pair ?f) (function-pair ?g)]))
        (= (function0 (composition [?f ?g]))
            (type.ftn:composition [(function0 ?f) (function0 ?g)]))
        (= (function1 (composition [?f ?g]))
            (type.ftn:composition [(function1 ?f) (function1 ?g)]))))))
(forall ((parallel-pair-morphism ?h)
         (exists ((parallel-pair-morphism ?f) (parallel-pair-morphism ?g)
                 (composable-pair [?f ?g]))
           (= (composition [?f ?g]) ?h)))
  (forall ((parallel-pair-morphism ?f) (parallel-pair-morphism ?g)
           (parallel-pair-morphism ?h)
           (composable-pair [?f ?g]) (composable-pair [?g ?h]))
    (= (composition [?f (composition [?g ?h])])
       (composition [(composition [?f ?g]) ?h])))

```

For any parallel pair $X = x_0, x_1 : X_0 \rightarrow X_1$, there is an *identity* parallel pair morphism $1_X : X \rightarrow X$ whose components are defined factorwise: $(1_X)_0 = 1_{X_0}$ and $(1_X)_1 = 1_{X_1}$. Identity is injective; hence, opspans can be regarded as special opspan morphisms that satisfy the unit laws. Identity satisfies two unit laws with respect to composition: composition with the identity of the source (target) of a opspan morphism returns that opspan morphism.

```

(iff:function identity)
(= (iff:source identity) type.dgm.ppr.obj:parallel-pair)
(= (iff:target identity) parallel-pair-morphism)
(forall ((type.dgm.ppr.obj:parallel-pair ?X)
         (and (= (source (identity ?X)) ?X)
              (= (target (identity ?X)) ?X)
              (= (function0 (identity ?X)) (type.ftn:identity (type.dgm.ppr.obj:set0 ?X)))
              (= (function1 (identity ?X)) (type.ftn:identity (type.dgm.ppr.obj:set1 ?X)))))
  (forall ((type.dgm.ospn.obj:parallel-pair ?X0) (type.dgm.ospn.obj:parallel-pair ?X1)
           (= (identity ?X0) (identity ?X1)))

```

```
(= ?X0 ?X1)

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

1.7.7 Type Opspans

type.dgm.ospn

Objects.

type.dgm.ospn.obj

An *opspan* is an object in the comma category

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

over the functorial ospan $1_{\mathbf{Set}^2} : \mathbf{Set}^2 \rightarrow \mathbf{Set}^2 \leftarrow \mathbf{Set} : \Delta$. 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, for simplicity, we use the second definition below (also defining the components), which is isomorphic.

$$\begin{aligned} \widehat{\mathbf{ospn}} &= \{((X_0, X_1), X, (x_0, x_1)) \mid \\ &\quad (X_0, X_1) \in \mathbf{set}^2, X \in \mathbf{set}, (x_0, x_1) \in \mathbf{ftn}^2, \\ &\quad \partial_0(x_0, x_1) = (X_0, X_1), \partial_1(x_0, x_1) = \Delta(X)\}, \text{ or} \\ \mathbf{ospn} &= \{(x_0, x_1) \mid x_0, x_1 \in \mathbf{ftn}, \partial_1(x_0) = \partial_1(x_1)\} \subseteq \mathbf{ftn} \times \mathbf{ftn}. \end{aligned}$$

The map $\widehat{\mathbf{ospn}} \rightarrow \mathbf{ospn}$ is just projection; the map $\mathbf{ospn} \rightarrow \widehat{\mathbf{ospn}}$ 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. The opspans that are axiomatized here are concrete: they can be referenced as

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

Here is the axiomatization.

```
(iff:set opspan)
(forall ((opspan ?x)) (type.dgm.pr.mor:function-pair ?x))
(forall ((type.ftn:function ?x0) (type.ftn:function ?x1))
  (<=> (opspan [?x0 ?x1])
    (= (type.ftn:target ?x0) (type.ftn:target ?x1))))

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

(iff:function function0) (iff:function opzeroth) (= opzeroth function0)
(= (iff:source function0) opspan)
(= (iff:target function0) type.ftn:function)
(forall ((opspan ?x))
  (= (function0 ?x) (type.dgm.pr.mor:function0 (function-pair ?x))))

(iff:function function1) (iff:function opfirst) (= opfirst function1)
```

```

(= (iff:source function1) opspan)
(= (iff:target function1) type.ftn:function)
(forall ((opspan ?x))
  (= (function1 ?x) (type.dgm.pr.mor:function1 (function-pair ?x))))

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

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

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

(iff:function set) (iff:function opvertex) (= opvertex set)
(= (iff:source set) opspan)
(= (iff:target set) type.set:set)
(forall ((opspan ?x))
  (and (= (set ?x) (type.ftn:target (function0 ?x)))
        (= (set ?x) (type.ftn:target (function1 ?x)))))

(forall ((opspan ?x))
  (= (type.dgm.pr.obj:constant (set ?x)) (type.ftn:target (function-pair ?x))))

```

Associated with any opspan $X = (x_0 : X_0 \rightarrow X_\bullet \leftarrow X_1 : x_1)$ is a *relation* $\mathbf{rel}(X) \subseteq X_0 \times X_1$, whose extent is defined to be the pullback set $\{(a_0, a_1) \mid x_0(a_0) = x_1(a_1)\}$ and whose projections are the pullback projections.

```

(iff:function relation)
(= (iff:source relation) opspan)
(= (iff:target relation) type.rel:relation)
(forall ((opspan ?os))
  (and (= (type.rel:set0 (relation ?os)) (set0 ?os))
        (= (type.rel:set1 (relation ?os)) (set1 ?os))
        (= (type.rel:extent (relation ?os)) (type.lim.pbk.obj:pullback ?os))
        (= (type.rel:projection0 (relation ?os)) (type.lim.pbk.obj:projection0 ?os))
        (= (type.rel:projection1 (relation ?os)) (type.lim.pbk.obj:projection1 ?os))))

```

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

```

(iff:function constant)
(= (iff:source constant) type.set:set)
(= (iff:target constant) opspan)
(forall ((type.set:set ?X))
  (and (= (set0 (constant ?X)) ?X)
        (= (set1 (constant ?X)) ?X)
        (= (set (constant ?X)) ?X)

```

```
(= (function0 (constant ?X)) (type.ftn:identity ?X))
(= (function1 (constant ?X)) (type.ftn:identity ?X)))
```

Any type *opspan* has an associated *parallel pair*, whose component functions are the composite of the product projections of the binary product of the pair of sets overlying the *opspan* with the component functions of the *opspan*. The equalizer and inclusion of this parallel pair can be used to define the pullback and pullback projections. This defines the parallel pair function $\text{obj}(\mathbf{E}) : \mathbf{ospn} \rightarrow \mathbf{ppr}$, which is the object function of the parallel pair functor $\mathbf{E} : \mathbf{Ospn} \rightarrow \mathbf{Ppr}$.

```
(iff:function parallel-pair)
(= (iff:source parallel-pair) opspan)
(= (iff:target parallel-pair) type.dgm.ppr.obj:parallel-pair)
(forall ((opspan ?os))
  (and (= (type.dgm.ppr.obj:set0 (parallel-pair ?os))
         (type.lim.prd2.obj:binary-product (set-pair ?os)))
       (= (type.dgm.ppr.obj:set1 (parallel-pair ?os)) (opvertex ?os))
       (= (type.dgm.ppr.obj:function0 (parallel-pair ?os))
          (type.ftn:composition
            [(type.lim.prd2.obj:projection0 (set-pair ?os)) (opzeroth ?os)]))
       (= (type.dgm.ppr.obj:function1 (parallel-pair ?os))
          (type.ftn:composition
            [(type.lim.prd2.obj:projection1 (set-pair ?os)) (opfirst ?os)]))))))
```

For any *opspan* $X = (x_0 : X_0 \rightarrow X_\bullet \leftarrow X_1 : x_1)$, there is an *opposite* *opspan* $X^{\text{op}} = (x_1 : X_1 \rightarrow X_\bullet \leftarrow X_0 : x_0)$. This defines the opposite function $\alpha : \mathbf{ospn} \rightarrow \mathbf{ospn}$, which is the object function of the involution functor $\alpha : \mathbf{Ospn}^{\text{op}} \rightarrow \mathbf{Ospn}$.

```
(iff:function opposite)
(= (iff:source opposite) opspan)
(= (iff:target opposite) opspan)
(forall ((opspan ?os))
  (and (= (opzeroth (opposite ?os)) (opfirst ?os))
       (= (opfirst (opposite ?os)) (opzeroth ?os))
       (= (opvertex (opposite ?os)) (opvertex ?os))
       (= (set0 (opposite ?os)) (set1 ?os))
       (= (set1 (opposite ?os)) (set0 ?os))))
```

The opposite of the opposite is the original *opspan*.

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

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

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

over the functorial *opspan* $1_{\mathbf{Set}^2} : \mathbf{Set}^2 \rightarrow \mathbf{Set}^2 \leftarrow \mathbf{Set} : \Delta$. 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 $((f_0, f_1), f_\bullet)$ consisting of a function pair $(f_0, f_1) : (X_0, X_1) \rightarrow (Y_0, Y_1)$ and an (opvertex) function $f_\bullet : X \rightarrow Y$, which satisfy the following commutativity condition in the category \mathbf{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 $\mathbf{ospn}\text{-mor} \subseteq \mathbf{ospn} \times \mathbf{ftn}^2 \times \mathbf{ftn} \times \mathbf{ospn}$ denote the collection of opspan morphisms. We name the projections

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

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

```

(iff:set opspan-morphism)
(forall ((opspan-morphism ?om))
  (exists ((type.dgm.ospn.obj:opspan ?X)
    (type.dgm.pr.mor:function-pair ?f01) (type.set:function ?f)
    (type.dgm.ospn.obj:opspan ?Y))
    (= ?om [?X ?f01 ?f ?Y])))
(forall ((type.dgm.ospn.obj:opspan ?X)
  (type.dgm.pr.mor:function-pair ?f01) (type.set:function ?f)
  (type.dgm.ospn.obj:opspan ?Y))
  (<=> (opspan-morphism [?X ?f01 ?f ?Y])
    (= (type.dgm.pr.mor:composition [?f01 (type.dgm.ospn.obj:function-pair ?Y)])
      (type.dgm.pr.mor:composition [(type.dgm.ospn.obj:function-pair ?X)
        (type.dgm.pr.mor:constant ?f)]))))

(iff:function function-pair)
(= (iff:source function-pair) opspan-morphism)
(= (iff:target function-pair) type.dgm.pr.mor:function-pair)
(forall ((type.dgm.ospn.obj:opspan ?X)
  (type.dgm.pr.mor:function-pair ?f01) (type.set:function ?f)
  (type.dgm.ospn.obj:opspan ?Y)
  (opspan-morphism [?X ?f01 ?f ?Y]))
  (= (function-pair [?X ?f01 ?f ?Y]) ?f01))

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

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

(iff:function function) (iff:function opvertex) (= opvertex function)
(= (iff:source function) opspan-morphism)
(= (iff:target function) type.ftn:function)
(forall ((type.dgm.ospn.obj:opspan ?X)

```

```

      (type.dgm.pr.mor:function-pair ?f01) (type.set:function ?f)
      (type.dgm.ospn.obj:opspan ?Y)
      (opspan-morphism [?X ?f01 ?f ?Y]))
    (= (function [?X ?f01 ?f ?Y]) ?f))

(iff:function source)
(= (iff:source source) opspan-morphism)
(= (iff:target source) type.dgm.ospn.obj:opspan)
(forall ((type.dgm.ospn.obj:opspan ?X)
         (type.dgm.pr.mor:function-pair ?f01) (type.set:function ?f)
         (type.dgm.ospn.obj:opspan ?Y)
         (opspan-morphism [?X ?f01 ?f ?Y])))
  (= (source [?X ?f01 ?f ?Y]) ?X))

(iff:function target)
(= (iff:source target) opspan-morphism)
(= (iff:target target) type.dgm.ospn.obj:opspan)
(forall ((type.dgm.ospn.obj:opspan ?X)
         (type.dgm.pr.mor:function-pair ?f01) (type.set:function ?f)
         (type.dgm.ospn.obj:opspan ?Y)
         (opspan-morphism [?X ?f01 ?f ?Y])))
  (= (target [?X ?f01 ?f ?Y]) ?Y))

```

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

```

(iff:function constant)
(= (iff:source constant) type.ftn:function)
(= (iff:target constant) opspan-morphism)
(forall ((type.ftn:function ?f))
  (and (= (source (constant ?f)) (type.dgm.ospn.obj:constant (type.ftn:source ?f)))
        (= (target (constant ?f)) (type.dgm.ospn.obj:constant (type.ftn:target ?f)))
        (= (function-pair (constant ?f)) (type.dgm.pr.mor:constant ?f))
        (= (function (constant ?f)) ?f)))

```

Any type opspan morphism has an associated *parallel pair morphism*, whose first component function is the binary product of the pair of functions overlying the opspan morphism and whose second component function is the (opvertex) function underlying the opspan morphism. This defines the parallel pair function $\mathbf{mor}(\mathbf{E}) : \mathbf{ospn-mor} \rightarrow \mathbf{ppr-mor}$, which is the morphism function of the parallel pair functor $\mathbf{E} : \mathbf{Ospn} \rightarrow \mathbf{Ppr}$.

```

(iff:function parallel-pair-morphism)
(= (iff:source parallel-pair-morphism) opspan-morphism)
(= (iff:target parallel-pair-morphism) type.dgm.ppr.mor:parallel-pair-morphism)
(forall ((opspan-morphism ?f))
  (and (= (type.dgm.ppr.mor:source (parallel-pair-morphism ?f))
          (type.dgm.ppr.obj:parallel-pair (source ?f)))
        (= (type.dgm.ppr.mor:target (parallel-pair-morphism ?f))
          (type.dgm.ppr.obj:parallel-pair (target ?f)))
        (= (type.dgm.ppr.mor:function0 (parallel-pair-morphism ?f))
          (type.lim.prd2.mor:binary-product (function-pair ?f)))
        (= (type.dgm.ppr.mor:function1 (parallel-pair-morphism ?f)) (opvertex ?f))))

```

For any opspan morphism $f = ((f_0, f_1), f_\bullet) : X \rightarrow Y$ from opspan $X = (x_0 : X_0 \rightarrow X_\bullet \leftarrow X_1 : x_1)$ to opspan $Y = (y_0 : Y_0 \rightarrow Y_\bullet \leftarrow Y_1 : y_1)$, there is an *opposite* opspan morphism $f^{\text{op}} = ((f_1, f_0), f_\bullet) : X^{\text{op}} \rightarrow Y^{\text{op}}$. This defines the

opposite function $\alpha : \mathbf{ospn}\text{-mor} \rightarrow \mathbf{ospn}\text{-mor}$, which is the morphism function of the involution functor $\alpha : \mathbf{Ospn}^{\text{op}} \rightarrow \mathbf{Ospn}$.

```
(iff:function opposite)
(= (iff:source opposite) opspan-morphism)
(= (iff:target opposite) opspan-morphism)
(forall ((opspan-morphism ?f))
  (and (= (source (opposite ?f)) (type.dgm.ospn.obj:opposite (source ?f)))
        (= (target (opposite ?f)) (type.dgm.ospn.obj:opposite (target ?f)))
        (= (function-pair (opposite ?f)) (type.dgm.pr.mor:opposite (function-pair ?f)))
        (= (function (opposite ?f)) (function ?f))))
```

The opposite of the opposite is the original opspan.

```
(forall ((opspan-morphism ?f))
  (= (opposite (opposite ?f)) ?f))
```

Category Theory. Two type opspan morphisms are *composable* when the target of the first is equal to the source of the second. We name the extent and projection factors of the composable endorelation.

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

(iff:function factor0)
(= (iff:source factor0) composable-pair)
(= (iff:target factor0) opspan-morphism)
(forall ((opspan-morphism ?f) (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 ((opspan-morphism ?f) (opspan-morphism ?g) (composable-pair [?f ?g]))
  (= (factor1 [?f ?g]) ?g))
```

The *composition* of two composable type opspan morphisms is defined factor-wise. 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 ((opspan-morphism ?f) (opspan-morphism ?g) (composable-pair [?f ?g]))
  (and (= (source (composition [?f ?g])) (source ?f))
        (= (target (composition [?f ?g])) (target ?g))
        (= (function-pair (composition [?f ?g]))
            (type.dgm.pr.mor:composition [(function-pair ?f) (function-pair ?g)]))
        (= (function (composition [?f ?g]))
            (type.ftn:composition [(function ?f) (function ?g)]))))

(forall ((opspan-morphism ?h))
```

```

(exists ((opspan-morphism ?f) (opspan-morphism ?g) (composable-pair [?f ?g]))
  (= (composition [?f ?g]) ?h))

(forall ((opspan-morphism ?f) (opspan-morphism ?g) (opspan-morphism ?h)
  (composable-pair [?f ?g]) (composable-pair [?g ?h]))
  (= (composition [?f (composition [?g ?h])])
    (composition [(composition [?f ?g]) ?h])))

```

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

```

(iff:function identity)
(= (iff:source identity) type.dgm.ospn.obj:opspan)
(= (iff:target identity) opspan-morphism)
(forall ((type.dgm.ospn.obj:opspan ?o))
  (and (= (source (identity ?o)) ?o)
    (= (target (identity ?o)) ?o)
    (= (function-pair (identity ?o))
      (type.dgm.pr.mor:identity (type.dgm.ospn.obj:set-pair ?o)))
    (= (function (identity ?o))
      (type.ftn:identity (type.dgm.ospn.obj:set ?o))))))

(forall ((type.dgm.ospn.obj:opspan ?o0) (type.dgm.ospn.obj:opspan ?o1))
  (= (identity ?o0) (identity ?o1)))
(= ?o0 ?o1))

(forall ((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.