

The IFF Type Namespace

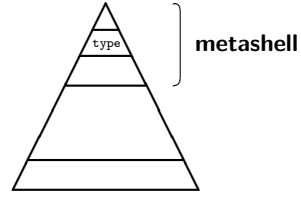
Robert E. Kent

December 18, 2007

Contents

1	The Type Kernel	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

Namespace Prefix
Technical: type
Recommended: type

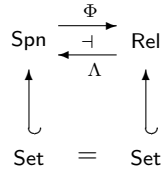


1 The Type Kernel

1.1 Introduction

The type namespace, a small-sized namespace at the top of the IFF architecture, is at the heart of the IFF metashell. Since the metashell unfolds into the rest of the IFF, we could regard the type namespace, and its kernel in particular, as the heart of the entire IFF. The type namespace is in the middle of the metashell, just below the IFF namespace. But, since the IFF namespace only specifies the set and function collections, it doesn't contain much axiomatization. The type namespace is where the axiomatization really begins. It has three nested subnamespaces: the type kernel and namespaces for type diagrams and type limits.

The type namespace kernel defines the finitely-complete category $\mathbf{Set} = \mathbf{Set}_{\text{type}}$ of Cantorian featureless sets and functions enriched with factorization and subobjects. In a standard fashion, the finitely-complete category \mathbf{Set} defines the bicategory $\mathbf{Span} = \mathbf{spn}(\mathbf{Set})$ of sets and spans, the ordered category $\mathbf{Rel} = \mathbf{rel}(\mathbf{Set})$ of sets and (binary) relations, and their connections. The category \mathbf{Set} embeds into the other two, and there is an adjunctive structure $\Phi \dashv \Lambda$ linking the embedding Λ and flattening Φ passages between \mathbf{Span} and \mathbf{Rel} : any relation is a span and any span flattens to a relation. The type kernel implicitly specifies these categorical, functorial and adjunctive structures. In addition to canonical functionality for finite limits, \mathbf{Set} has factorizations and subobjects.



There are two methods of factorization that are isomorphic to each other: functions factor through their range set and through the quotient of their kernel equivalence relation. Factorization provides the unit for the adjunctive structure $\Phi \dashv \Lambda$.

The type namespace kernel defines four related subobject structures (Table 1). For any set X , the kernel specifies (upper left) the slice category of functions over (generalized elements in) X , which collapses to the belonging order over X , and (upper right) the subset order on predicates (parts) with genus X . For any set pair (X_0, X_1) , the kernel specifies (lower left) the category of spans over (generalized element pairs in) (X_0, X_1) , which collapses to the belonging order over (X_0, X_1) , and (lower right) the subrelation order on

relations with set pair (X_0, X_1) . There are membership relations between functions and predicates and between spans and relations. Associated with these two membership relations are maps that return the proof of membership.

There could be various transformation maps possible on a variety of objects. Following the principle of conceptual warrant, we will specify whatever we need and no more. There can be

{direct, inverse} image transforms of
 a {subset, predicate, endorelation}
 along a {function, multivalued-function, relation},

ostensively offering $18 = 2 \times 3 \times 3$ possibilities. However, subsets are isomorphic to predicates and multivalued functions are isomorphic to relations, thus conceptually reducing these possibilities to $8 = 2 \times 2 \times 2$. Moreover, in a sense an endorelation is a predicate on a power. Hence, it is conceptually possible that we can reduce this number to $4 = 2 \times 1 \times 2$. In the current type namespace we have specified the direct (or power) and inverse image operations of subsets along functions, multivalued-functions (conceptually equivalent to the relations) and relations. However, at some point we may need the direct universal image \forall_f for functions f , the right adjoint in the chain $\exists_f \dashv (-)_f^{-1} \dashv \forall_f$, where direct (existential) image is symbolized by \exists_f and inverse image or substitution is symbolized by $(-)_f^{-1}$.

The meta namespace in the IFF metashell is the next level down from the type namespace. By and large, the meta namespace is the specialization of the type namespace. The important and necessary topics on which the meta namespace differs from (and adds to) the type namespace are listed here. These cannot be axiomatized in the type namespace because of size considerations.

Constant Functions: Although the type namespace axiomatizes constant functions, it cannot axiomatize their parametric aspect.

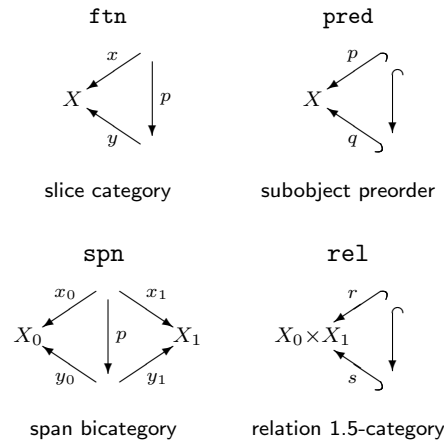
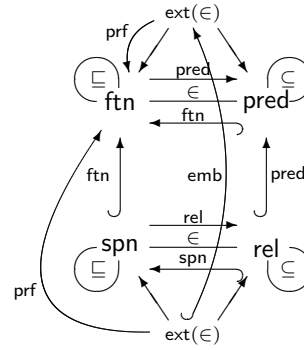
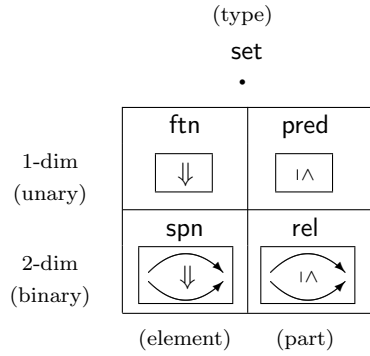
Exponents: Exponents, curry functions, and their adjunction characterization (with respect to products).

External Transformations: These consist of existential/universal quantification and inverse image functors between slice categories.

Subobjects: Complete functionality for subobjects can only be defined for the “quotient” case of ordinary subsets; in the type namespace we can only provide define the membership and inclusion relations for predicates and relations. In general, although the collection of all subsets is locally small, the collection of all subobjects is not locally small. This corresponds to, and in a sense is derivative from, the incomplete functionality for slice categories for functions and spans.

Parametric Composition: This is probably the most important topic in the IFF core.

Like other parts of the IFF metashell, the type namespace is closely related to the IFF grammar (see the document entitled “The IFF Syntax”), which



Concept	IFF Term
$\text{pred} : \text{ftn} \rightarrow \text{pred}$	predicate
$\text{ftn} : \text{pred} \rightarrow \text{ftn}$	function
$\text{rel} : \text{spn} \rightarrow \text{rel}$	relation
$\text{spn} : \text{rel} \rightarrow \text{spn}$	span
$\text{ftn} : \text{spn} \rightarrow \text{ftn}$	function
$\text{pred} : \text{rel} \rightarrow \text{pred}$	predicate

ftn-pred reflection:
 $f \xrightarrow{\eta} \text{ftn}(\text{pred}(f)), \quad \forall f \in \text{ftn}$
 $p \cong \text{pred}(\text{ftn}(p)), \quad \forall p \in \text{pred}$

spn-rel reflection:
 $s \xrightarrow{\eta} \text{spn}(\text{rel}(s)), \quad \forall s \in \text{spn}$
 $r \cong \text{rel}(\text{spn}(r)), \quad \forall r \in \text{rel}$

pred-sub equivalence:
 $XY = \text{sub}(\text{pred}(XY)), \quad \forall XY \in \text{sub}$
 $p \cong \text{pred}(\text{sub}(p)), \quad \forall p \in \text{pred}$

Table 1: Subobject Architecture

	declaration	syntactic construct
primitive	(iff:set X)	(X x)
	(iff:function f)	(f x)
	(type.spn:span x)	...
defined	(type.pred:predicate p)	(p x)
	(type.rel:relation r)	(r x y)

Table 2: IFF Atomic Expression

specifies the correct form for IFF expressions. Atomic expression in the IFF syntax, which is represented in prefix notation, consists of four syntactic or type-theoretic constructs (Table 2): set membership, predicate invocation, function application and relation invocation¹. Set membership and function application, which are regarded as distinct primitive notions in the IFF, are handled at the highest level in the iff namespace. Predicate and relation invocation, which are defined notions in the IFF, are handled at the intermediate level in the type namespace. The IFF grammar can and should be used to check for type correctness and well-formed-ness for all four of these type-theoretic constructs: an error should be signaled, (1) if a set or predicate symbol is being used as a function (in a term) or a relation (in a relational expression), (2) if a function symbol is being used as a set or predicate (in a declaration), a set component for functions and relations or as a relation (in a relational expression), and (3) if a relation symbol is being used as a set or predicate (in a declaration), a set component for functions and relations or as a function (in a term).

The terminology for the kernel of the type namespace, is listed in Table 3, Table 4, Table 5 and Table 6. This consists of 224 terms representing 213 concepts (with 11 synonyms). There are five basic collections: the collection **set** of type *sets*, the collection **ftn** of type *functions*, the collection **spn** of type *spans*, the collection **pred** of type *predicates*, and the collection **rel** of (binary) type *relations*. These collections are distinct and pairwise disjoint. Some of the components of these four basic collections are also introduced here. There is a *genus* map from predicates to sets, there are *source* and *target* maps from functions to sets, and there is a domain or *zeroth* map and a codomain or *first* map from relations to sets.

¹The IFF does not use an atomic syntactic construct for span application. For a span $x = (x_0 : X_0 \leftarrow X \rightarrow X_1 : x_1)$, application would consist of inputting an element $a \in X$ and outputting a pair $(x_0(a), x_1(a)) \in X_0 \times X_1$. Span application, which should resemble the form of function application $(x \ a)$, could be defined by $(= (x \ a) [(x_0 \ a) (x_1 \ a)])$.

(Emphasized terms are IFF terms.)

IFF Set IFF Function

.set	
<i>basics</i>	<i>set</i> isomorphic-extent [in]finite-differentia
<i>instances</i>	zero = initial = empty one = terminal two three
<i>subobject</i>	subordinate
	smaller larger inclusion predicate constant-one power power-pair leq bottom top binary-meet binary-join minus complement implication singleton union = join intersection = meet exists = power inverse-image forall
.ftn	
<i>basics</i>	<i>function</i> [optimal-]restriction-extent
<i>category theory</i>	<i>source target</i> [optimal-](smaller larger)
<i>factorization</i>	element constant-function
	set constant-element construction
<i>conversion</i>	composable-pair
<i>instances</i>	appliable-pair
.ftn.mor	range kernel coimage
<i>basics</i>	injection surjection bijection
	injective-factor coapplication surjective-factor coequalizer inverse
<i>category theory</i>	predicate span relation fiber unit
	initial terminal counique unique
<i>basics</i>	2-cell belonging equivalence isomorphism
<i>category theory</i>	source target function element0 element1
	composable-pair
	factor0 factor1 composition identity

Technical Prefix : type.krnl
Recommended Prefix : type

Table 3: The Kernel Type Namespace (set/function)

(Emphasized terms are IFF terms.)

	IFF Set	IFF Function
.spn		
<i>basics</i>	span	set = vertex set0 set1 set-pair function0 function1 function-pair smaller larger
<i>category theory</i>	composable-pair	factor0 factor1 opspan composition identity
<i>factorization</i>	combinable-pair	component0 component1 combination
	injection surjection bijection	extent = range kernel coimage injective-factor coapplication surjective-factor coequalizer
<i>conversion instances</i>		opposite function relation unit initial terminal counique unique
.spn.mor		
<i>basics</i>	2-cell belonging equivalence isomorphism	source target function = vertex set0 set1 element0 element1
<i>conversion</i>		function-2-cell
.spn.mor.vrt		
<i>vertical category theory</i>	composable-pair	factor0 factor1 composition identity
.spn.mor.hrz		
<i>horizontal category theory</i>	composable-pair	factor0 factor1 composition identity opspan-morphism

Technical Prefix : type.krnl

Recommended Prefix : type

Table 4: The Kernel Type Namespace (span)

	IFF Set	IFF Function
.pred		
<i>basics</i>	predicate delimitation-extent	genus differentia smaller larger
<i>conversion</i>		function subordinate injection
<i>subobject</i>	inclusion-extent equivalence = isomorphism membership	part0 part1 element part proof

Technical Prefix : type.krnl

Recommended Prefix : type

Table 5: The Kernel Type Namespace (predicate)

IFF Set

IFF Function

<code>.rel</code>																	
<i>basics</i>	<table border="1"> <tr> <td><code>relation</code></td> <td><code>extent set0 set1 set-pair projection0 projection1 smaller larger</code></td> </tr> <tr> <td><i>category theory</i></td> <td><code>composable-pair</code></td> </tr> <tr> <td><i>conversion</i></td> <td><code>function predicate span injection opposite fiber01 fiber10</code></td> </tr> </table>	<code>relation</code>	<code>extent set0 set1 set-pair projection0 projection1 smaller larger</code>	<i>category theory</i>	<code>composable-pair</code>	<i>conversion</i>	<code>function predicate span injection opposite fiber01 fiber10</code>										
<code>relation</code>	<code>extent set0 set1 set-pair projection0 projection1 smaller larger</code>																
<i>category theory</i>	<code>composable-pair</code>																
<i>conversion</i>	<code>function predicate span injection opposite fiber01 fiber10</code>																
<i>subobject</i>	<table border="1"> <tr> <td><code>inclusion-extent</code></td> <td><code>part0 part1</code></td> </tr> <tr> <td><code>equivalence = isomorphism</code></td> <td></td> </tr> <tr> <td><code>membership</code></td> <td><code>element part proof</code></td> </tr> </table>	<code>inclusion-extent</code>	<code>part0 part1</code>	<code>equivalence = isomorphism</code>		<code>membership</code>	<code>element part proof</code>										
<code>inclusion-extent</code>	<code>part0 part1</code>																
<code>equivalence = isomorphism</code>																	
<code>membership</code>	<code>element part proof</code>																
<i>basics</i> <i>order theory</i>	<table border="1"> <tr> <td><code>endorelation</code></td> <td><code>set</code></td> </tr> <tr> <td><code>reflexive-relation</code></td> <td></td> </tr> <tr> <td><code>transitive-relation</code></td> <td></td> </tr> <tr> <td><code>antisymmetric-relation</code></td> <td></td> </tr> <tr> <td><code>symmetric-relation</code></td> <td></td> </tr> <tr> <td><code>preorder partial-order</code></td> <td></td> </tr> <tr> <td><code>equivalence-relation</code></td> <td><code>quotient canon</code></td> </tr> <tr> <td><code>respects-extent</code></td> <td></td> </tr> </table>	<code>endorelation</code>	<code>set</code>	<code>reflexive-relation</code>		<code>transitive-relation</code>		<code>antisymmetric-relation</code>		<code>symmetric-relation</code>		<code>preorder partial-order</code>		<code>equivalence-relation</code>	<code>quotient canon</code>	<code>respects-extent</code>	
<code>endorelation</code>	<code>set</code>																
<code>reflexive-relation</code>																	
<code>transitive-relation</code>																	
<code>antisymmetric-relation</code>																	
<code>symmetric-relation</code>																	
<code>preorder partial-order</code>																	
<code>equivalence-relation</code>	<code>quotient canon</code>																
<code>respects-extent</code>																	

Technical Prefix : `type.krn1`
 Recommended Prefix : `type`

Table 6: The Kernel Type Namespace (relation)

1.2 Type Sets

Basics. There is an IFF set `set` of all type *sets*. Any type set is itself an IFF set; that is, the IFF set of all type sets is an (implicit) subset of the set of all IFF sets.

```
(iff:set set)
(forall ((set ?X)) (iff:set ?X))
```

There is a binary *isomorphic* endorelation between pairs of type sets, that are linked by an bijection². Since there is no notion of binary relation specified at the IFF level, at the type level we use the extent set of the isomorphism relation³ to indirectly specify it. The isomorphic endorelation is an equivalence relation (reflexive, symmetric and transitive), since identities are bijections, bijections have an inverse and bijections are closed under composition.

```
(iff:set isomorphic-relation)
(forall ((isomorphic-relation ?XY)) (type.dgm.pr.obj:set-pair ?XY))
(forall ((set ?X) (set ?Y))
  (<=> (isomorphic-relation [?X ?Y])
    (exists ((type.ftn:bijection ?f)
      (and (= ?X (type.ftn:source ?f)) (= ?Y (type.ftn:target ?f))))))
(forall ((set ?X))
  (isomorphic-relation [?X ?X]))
(forall ((set ?X) (set ?Y))
  (=> (isomorphic-relation [?X ?Y])
    (isomorphic-relation [?Y ?X])))
(forall ((set ?X) (set ?Y) (set ?Z))
  (=> (and (isomorphic-relation [?X ?Y]) (isomorphic-relation [?Y ?Z]))
    (isomorphic-relation [?X ?Z])))
```

The following is taken from the book *Sets for Mathematics* (2003) by Lawvere and Rosebrugh.

Fact 1 (Galileo) *The set $\text{Natno} = \mathbb{N} = \{0, 1, 2, \dots\}$ of all natural numbers is isomorphic to the set $\text{Sqr} = \{0, 2, 4, \dots\} \subset \text{Natno}$ of all square whole numbers (a proper subset).*

Proof. Use the squaring function $(-)^2 : \text{Natno} \xrightarrow{\sim} \text{Sqr}$ and its inverse the square root function $\sqrt{} : \text{Sqr} \xrightarrow{\sim} \text{Natno}$. ■

This observation by Galileo was generalized into a definition by Dedekind.

Definition 1 (Dedekind) *A set X is finite when all injections on X are bijections. A set X is infinite when there is at least one injection on X that is not surjective.*

²Here we follow and quote from the discussion on the topic of isomorphism and Dedekind finiteness as presented in Lawvere and Rosebrugh [1]. “The notation $f : X \xrightarrow{\sim} Y$ means that f is an isomorphism. One type set X is *isomorphic* to a type set Y when there is at least one isomorphism (type bijection) from X to Y . This definition of isomorphism is used in all categories, but in a category of abstract sets and arbitrary functions the two type sets X and Y are said to be *equinumerous* or to *have the same cardinality*.” As Lawvere and Rosebrugh point out, the isomorphism of abstract sets offers a method to study equinumerosity without counting, a fact systematically used by Cantor.

³We use this idea of “extent serving as proxy” for other relations and orders, such as subset, delimitation, (optimal-)restriction, abridgment, function (element) belonging, predicate (part) inclusion and the member relations between functions and predicates and between spans and relations. This is a small part of the bootstrapping mechanism of the IFF metashell.

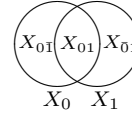
These are predicates (adjectives) that refer to (modify) sets (nouns), the genus, and result in subsets (noun phrases), the differentia.

```
(iff:set finite-set)
(forall ((finite-set ?X)) (set ?X))
(forall ((set ?X))
  (<=> (finite-set ?X)
    (forall ((type.ftn:injection ?f)
      (= (type.ftn:source ?f) ?X)
      (= (type.ftn:target ?f) ?X))
      (type.ftn:bijection ?f))))

(iff:set infinite-set)
(forall ((infinite-set ?X)) (set ?X))
(forall ((set ?X))
  (<=> (infinite-set ?X)
    (not (finite-set ?X))))
```

For any pair of type sets X_0 and X_1 , there are *binary union*, *binary intersection* and (binary) *difference* type sets defined as follows.

$$\begin{aligned} X_0 \cup X_1 &= \{y \mid y \in X_0 \text{ or } y \in X_1\} = X_{0\bar{1}} \cup X_{01} \cup X_{\bar{0}1} \\ X_0 \cap X_1 &= \{y \mid y \in X_0 \text{ and } y \in X_1\} = X_{01} \\ X_0 \setminus X_1 &= \{y \mid y \in X_0 \text{ and } y \notin X_1\} = X_{0\bar{1}} \end{aligned}$$



```
(iff:function binary-union)
(= (iff:source binary-union) type.dgm.pr.obj:set-pair)
(= (iff:target binary-union) set)
(forall ((set ?X0) (set ?X1))
  (and (subset-relation [(binary-union [?X0 ?X1])] ?X0)
    (subset-relation [(binary-union [?X0 ?X1])] ?X1)
    (forall (((binary-union [?X0 ?X1]) ?x)
      (or (?X0 ?x) (?X1 ?x)))))

(iff:function binary-intersection)
(= (iff:source binary-intersection) type.dgm.pr.obj:set-pair)
(= (iff:target binary-intersection) set)
(forall ((set ?X0) (set ?X1))
  (and (subset-relation [(binary-intersection [?X0 ?X1]) ?X0]
    (subset-relation [(binary-intersection [?X0 ?X1]) ?X1])))
(forall ((set ?X0) (set ?X1) (?X0 ?x) (?X1 ?x))
  ((binary-intersection [?X0 ?X1]) ?x))

(iff:function difference)
(= (iff:source difference) type.dgm.pr.obj:set-pair)
(= (iff:target difference) set)
(forall ((set ?X0) (set ?X1))
  (and (subset-relation [(difference [?X0 ?X1]) ?X0]
    (forall (((difference [?X0 ?X1]) ?x)
      (not (?X1 ?x))
      (forall ((?X0 ?x) (not (?X1 ?x)))
        ((difference [?X0 ?X1]) ?x)))))
```

Here are a few of the many properties that relate these operations:

$$\begin{aligned}
X_0 \cup X_1 &= X_1 \cup X_0 \\
(X_0 \cup X_1) \cup X_2 &= X_0 \cup (X_1 \cup X_2) \\
X_0 \cap X_1 &= X_1 \cap X_0 \\
(X_0 \cap X_1) \cap X_2 &= X_0 \cap (X_1 \cup X_2) \\
(X_0 \setminus X_1) \cup X_1 &= X_0 \cup X_1 \\
(X_0 \setminus X_1) \cap X_1 &= \emptyset
\end{aligned}$$

$$\begin{aligned}
&\text{if } X \subseteq Y, \\
&\text{then } X \cup Y = Y, X \cap Y = X \text{ and } X \setminus Y = \emptyset \\
&\text{hence, } X \cup X = X, X \cup \emptyset = X, \\
&\quad X \cap X = X, X \cap \emptyset = \emptyset, \\
&\quad X \setminus \emptyset = X, X \setminus X = \emptyset
\end{aligned}$$

```

(forall ((set ?X0) (set ?X1))
  (and (= (binary-union [?X0 ?X1]) (binary-union [?X1 ?X0]))
        (= (binary-intersection [?X0 ?X1]) (binary-intersection [?X1 ?X0]))
        (= (binary-union [(difference [?X0 ?X1]) ?X1]) ?X1)
        (= (binary-intersection [(difference [?X0 ?X1]) ?X1]) zero)))

(forall ((set ?X0) (set ?X1) (set ?X2))
  (and (= (binary-union [(binary-union [?X0 ?X1]) ?X2])
        (binary-union [?X0 (binary-union [?X1 ?X2])]))
        (= (binary-intersection [(binary-intersection [?X0 ?X1]) ?X2])
        (binary-intersection [?X0 (binary-intersection [?X1 ?X2])]))))

(forall ((set ?X0) (set ?X1))
  (=> (subset-relation [?X0 ?X1])
      (and (= (binary-union [?X0 ?X1]) ?X1)
            (= (binary-intersection [?X0 ?X1]) ?X0)
            (= (difference [?X0 ?X1]) empty))))

(forall ((set ?X))
  (and (= (binary-union [?X ?X]) ?X) (= (binary-union [?X empty]) ?X)
        (= (binary-intersection [?X ?X]) ?X) (= (binary-intersection [?X empty]) empty)
        (= (difference [?X empty]) ?X) (= (difference [?X ?X]) empty)))

```

Instances. Here are some basic sets: **0** = zero = \emptyset , the initial empty set; **1** = one, the terminal set with one element; **2** = two = **1** + **1** and **3** = three = **1** + **1** + **1**. zero and one have several synonyms. two and three are often used for indexing. Nothing is in zero. The canonical object in one is denoted $0 \in \text{one}$. The canonical object in two, but not one, is denoted $1 \in \text{two} \setminus \text{one}$. The canonical object in three, but not two, is denoted $2 \in \text{three} \setminus \text{two}$. These three canonical objects are distinct: $0 \neq 1$, $1 \neq 2$ and $0 \neq 2$. **0** has the universal property that for any set X there is only one function $\mathbf{0} \rightarrow X$, and **1** has the universal property that for any set X there is only one function $X \rightarrow \mathbf{1}$. There is a *constant zero* function $\Delta_{\text{zero}} : \text{set} \xrightarrow{\text{!set}} \mathbf{1} \xrightarrow{\text{zero}} \text{set}$, which maps any meta set X to the meta set zero = 0. There is a *constant one* function $\Delta_{\text{one}} : \text{set} \xrightarrow{\text{!set}} \mathbf{1} \xrightarrow{\text{one}} \text{set}$, which maps any meta set X to the meta set one = 1. For any type set X , there is a *counique* type function $\text{!}_X : \emptyset \rightarrow X$ and a *unique* type function $\text{!}_X : X \rightarrow \mathbf{1}$. These are the unique functions between their respective sources and targets.

```

(iff:thing iff:0) (iff:thing iff:1) (iff:thing iff:2)
(not (= iff:0 iff:1)) (not (= iff:1 iff:2)) (not (= iff:0 iff:2))

```

```

(set zero) (set initial) (= initial zero) (set empty) (= empty initial)
(forall ((zero ?x)) (not (zero ?x)))

```

```

(set one) (set terminal) (= terminal one) (one 0)
(forall ((one ?x)) (= ?x 0))

(set two) (two 0) (two 1)
(forall ((two ?x)) (or (= ?x 0) (= ?x 1)))

(set three) (three 0) (three 1) (three 2)
(forall ((three ?x)) (or (= ?x 0) (= ?x 1) (= ?x 2)))

(iff:function constant-zero)
(= (iff:source constant-zero) set)
(= (iff:target constant-zero) set)
(forall ((set ?X))
  (= (constant-zero ?X) zero))

(iff:function constant-one)
(= (iff:source constant-one) set)
(= (iff:target constant-one) set)
(forall ((set ?X))
  (= (constant-one ?X) one))

(iff:function counique)
(= (iff:source counique) set)
(= (iff:target counique) type.ftn:function)
(forall ((set ?X))
  (and (= (type.ftn:source (counique ?X)) zero)
        (= (type.ftn:target (counique ?X)) ?X)
        (forall ((type.function ?f)
                  (= (type.ftn:source ?f) zero)
                  (= (type.ftn:target ?f) ?X))
          (= ?f (counique ?X)))))

(iff:function unique)
(= (iff:source unique) set)
(= (iff:target unique) type.ftn:function)
(forall ((set ?X))
  (and (= (type.ftn:source (unique ?X)) ?X)
        (= (type.ftn:target (unique ?X)) one)
        (forall ((type.function ?f)
                  (= (type.ftn:source ?f) ?X)
                  (= (type.ftn:target ?f) one))
          (= ?f (unique ?X)))))

```

The following inclusions (subset relationships) hold for instances: $\text{zero} \subseteq X$ for any set X , and $\text{one} \subset \text{two} \subset \text{three}$.

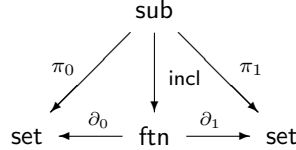
```

(forall ((set ?X)) (subset-relation [zero ?X]))
(subset-relation [one two]) (not (= one two))
(subset-relation [two three]) (not (= two three))

```

Subobject. There is a binary *subset* relation \subseteq on type sets. In the subset relationship $Y \subseteq X$, all elements of the smaller type set Y are members of the larger type set X . A pair of type sets is subordinate when it satisfies the subset relation. Let $\text{sub} = \text{ext}(\subseteq)$ denote the set of type subordinate pairs. We name the components of a subset relationship. There are projections $\pi_0^{\subseteq} : \text{sub} \rightarrow \text{set}$

and $\pi_1^{\subseteq} : \text{sub} \rightarrow \text{set}$ to the *smaller* and *larger* components. For each subordinate pair of type sets $Y \subseteq X$, there is an *inclusion* injection $\iota_{Y,X} = \text{incl}_{Y,X} : Y \hookrightarrow X : x \mapsto x$, whose source is the smaller type set and whose target is the larger type set. The subset relation on type sets is a partial order (reflexive, antisymmetric and transitive), since identities are inclusions and inclusions are closed under composition⁴.



```

(iff:set subset-relation)
(forall ((subset-relation ?YX)) (type.dgm.pr.obj:set-pair ?YX))
(forall ((set ?Y) (set ?X))
  (<=> (subset-relation [?Y ?X])
    (forall ((?Y ?x) (?X ?x))))

(iff:function smaller)
(= (iff:source smaller) subset-relation)
(= (iff:target smaller) set)
(forall ((set ?Y) (set ?X) (subset-relation [?Y ?X]))
  (= (smaller [?Y ?X]) ?Y))

(iff:function larger)
(= (iff:source larger) subset-relation)
(= (iff:target larger) set)
(forall ((set ?Y) (set ?X) (subset-relation [?Y ?X]))
  (= (larger [?Y ?X]) ?X))

(iff:function inclusion)
(= (iff:source inclusion) subset-relation)
(= (iff:target inclusion) type.ftn:function)
(forall ((set ?Y) (set ?X) (subset-relation [?Y ?X]))
  (and (= (type.ftn:source (inclusion [?Y ?X])) ?Y)
    (= (type.ftn:target (inclusion [?Y ?X])) ?X)
    (forall ((?Y ?y) (?X ?x))
      (= ((inclusion [?Y ?X]) ?y) ?x))))

(forall ((set ?X))
  (subset-relation [?X ?X]))
(forall ((set ?Y) (set ?X))
  (=> (and (subset-relation [?Y ?X]) (subset-relation [?X ?Y]))
    (= ?Y ?X)))

```

⁴We might be interested in extending the subset relation to a subobject relation. If we say that Y is a subobject of X , we mean that there is an injection $Y \hookrightarrow X$. There could be more than one of these. However, there is at least one and we could choose and name a canonical one of these. With this assumption, there would be a function $\text{inj} : \text{sub} \rightarrow \text{ftn}$ such that $\text{inj}_{Y,X} : Y \hookrightarrow X$ is an injection for any subordinate pair (Y, X) . Furthermore, we could assume that the chosen injection is the inclusion when $Y \subseteq X$. This would allow us to conservatively extend the delimitation, restriction and abridgment relations. The main problem is that the chosen injections do not strictly obey transitivity; that is, for subordinate pairs (Z, Y) and (Y, X) , we would have an isomorphism $\text{inj}_{Z,Y} \cdot \text{inj}_{Y,X} \cong \text{inj}_{Z,X}$, but not necessarily an identity. And we need an identity, to show transitivity for delimitation, restriction and abridgment.

```
(forall ((set ?Z) (set ?Y) (set ?X))
  (=> (and (subset-relation [?Z ?Y]) (subset-relation [?Y ?X]))
    (subset-relation [?Z ?X])))
```

The subset order can be defined in terms of Boolean operations.

$$\begin{aligned} Y \subseteq X &\text{ iff } Y \cap X = Y \\ &\text{ iff } Y \cup X = X \\ &\text{ iff } Y \setminus X = \emptyset \end{aligned}$$

```
(forall ((set ?Y) (set ?X))
  (<=> (subset-relation [?Y ?X])
    (= (binary-intersection [?Y ?X]) ?Y)))
```

```
(forall ((set ?Y) (set ?X))
  (<=> (subset-relation [?Y ?X])
    (= (binary-union [?Y ?X]) ?X)))
```

```
(forall ((set ?Y) (set ?X))
  (<=> (subset-relation [?Y ?X])
    (= (difference [?Y ?X]) empty)))
```

For every subordinate pair $Y \subseteq X$, there is a type *predicate* $\text{pred}_{Y,X} : Y \hookrightarrow X$, whose genus is X , whose differentia is Y and whose function is the inclusion $\text{incl}_{Y,X} : Y \hookrightarrow X$.

```
(iff:function predicate)
(= (iff:source predicate) subordinate)
(= (iff:target predicate) predicate)
(forall ((subordinate ?YX))
  (and (= (type.pred:genus (predicate ?YX)) (smaller ?YX))
    (= (type.pred:differentia (predicate ?YX)) (larger ?YX))
    (= (type.pred:function (predicate ?YX)) (inclusion ?YX))))
```

Any subordinate pair is identical to the subordinate pair of its predicate.

```
(forall ((subset-relation ?YX))
  (= ?YX (type.pred:subordinate (predicate ?YX))))
```

For any type set X , there is a *power* type set $\wp X$ consisting of the set of all subsets of X

$$\wp X = \{Y \in \text{set} \mid Y \subseteq X\}.$$

The IFF power function $\wp : \text{set} \rightarrow \text{set}$ is the (implicit) 10-fiber of the subset relation. It is an injection, since $\wp X_1 = \wp X_2$ implies $X_1 = \cup(\wp X_1) = \cup(\wp X_2) = X_2$ for any two sets $X_1, X_2 \in \text{set}$. For convenience, here we name the product of the pairing of the power function. For any type set X , a *power pair* is a pair of X -subsets $(Y_0, Y_1) \in \wp X^2 = \wp X \times \wp X$.

$$\begin{aligned} \wp X^0 &= \text{one} \\ \wp X^1 &= \wp X \\ \wp X^2 &= \wp X \times \wp X \end{aligned}$$

```

(iff:function power)
(= (iff:source power) set)
(= (iff:target power) set)
(forall ((set ?X) ((power ?X) ?Y)) (set ?Y))
(forall ((set ?X) (set ?Y))
  (<=> ((power ?X) ?Y)
    (subset-relation [?Y ?X])))

(iff:function power-pair)
(= (iff:source power-pair) set)
(= (iff:target power-pair) set)
(forall ((set ?X))
  (= (power-pair ?X) (type.lim.pwr2.obj:power (power ?X))))

```

Let X be any type set.

- The subset order on sets restricts to the power $\wp X$. There is a binary *leq* relationship \subseteq_X between pairs of X -subsets in $\wp X$. One X -subset $Y \subseteq X$ is less than or equal to another X -subset $Z \subseteq X$, denoted $Y \subseteq_X Z$, when Y is a subset of Z : $Y \subseteq_X Z$ iff $Y \subseteq Z$. This local inclusion relation is a partial order (reflexive, antisymmetric and transitive).
- There is a *bottom* X -subset $\lceil \perp_X \rceil : \Delta_{\text{one}}(X) = 1 \rightarrow \wp X$. The bottom X -subset \perp is the X -subset $\emptyset \subseteq X$ whose inclusion is the counique function $!_X : \emptyset \hookrightarrow X$; that is, for all elements $x \in X$, not $\perp(x)$.
- There is a *top* X -subset $\lceil \top_X \rceil : \Delta_{\text{one}}(X) = 1 \rightarrow \wp X$. The top X -subset \top is the X -subset $X \subseteq X$ whose inclusion is the identity $1_X : X \hookrightarrow X$; that is, for all elements $x \in X$, $\top(x)$.
- There is a *binary join (disjunction)* operation $\vee_X : \wp X \times \wp X \rightarrow \wp X$. For any two X -subset $Y \subseteq X$ and $Z \subseteq X$, the binary join is the X -subset $Y \vee Z = \{x \in X \mid x \in Y \text{ or } x \in Z\} \subseteq X$.
- There is a *binary meet (conjunction)* operation $\wedge_X : \wp X \times \wp X \rightarrow \wp X$. For any two X -subset $Y \subseteq X$ and $Z \subseteq X$, the binary meet is the X -subset $Y \wedge Z = \{x \in X \mid x \in Y \text{ and } x \in Z\} \subseteq X$.
- There is a *minus* operation $\setminus_X : \wp X \times \wp X \rightarrow \wp X$. For any two X -subsets $Y \subseteq X$ and $Z \subseteq X$, the minus is the X -subset $Y \setminus Z = \{x \in X \mid x \in Y \text{ or } x \notin Z\} \subseteq X$. Clearly, $Y \setminus Z = Y \wedge \neg Z$.
- There is a *complement (negation)* operation $\neg_X : \wp X \rightarrow \wp X$. For any X -subset $Y \subseteq X$, the complement is the X -subset $\neg Y = \{x \in X \mid x \notin Y\} \subseteq X$, the minus of the “universe” X with Y , $\neg Y = X \setminus Y$.
- There is a *implication* operation $\Rightarrow_X : \wp X \times \wp X \rightarrow \wp X$. For any two X -subsets $Y \subseteq X$ and $Z \subseteq X$, the implication is the X -subset $Y \Rightarrow Z = \{x \in X \mid x \in Y \text{ implies } x \in Z\} = \{x \in X \mid x \in Z \text{ or } x \notin Y\} \subseteq X$. Clearly, $Y \Rightarrow Z = \neg(Y \setminus Z) = \neg(Y \wedge \neg Z) = \neg Y \vee Z$.

```

(iff:function leq)
(= (iff:source leq) set)
(= (iff:target leq) type.rel:endorelation)
(forall ((set ?X))
  (and (= (type.rel:set (leq ?X)) (power ?X))
    (forall (((power ?X) ?Y) ((power ?X) ?Z))
      (<=> ((leq ?X) ?Y ?Z)
        (subset-relation [?Y ?Z])))))
(forall ((set ?X))
  (type.rel:partial-order (leq ?X)))
(forall ((set ?X) ((power ?X) ?Y) ((power ?X) ?Z))
  (<=> ((leq ?X) ?Y ?Z)
    (= ((binary-meet ?X) [?Y ?Z]) ?Y)))

(iff:function bottom)
(= (iff:source bottom) set)
(= (iff:target bottom) type.ftn:function)
(forall ((set ?X))
  (and (= (type.ftn:source (bottom ?X)) (constant-one ?X))
    (= (type.ftn:target (bottom ?X)) (power ?X))
    (= ((bottom ?X) iff:0) empty)))

(iff:function top)
(= (iff:source top) set)
(= (iff:target top) type.ftn:function)
(forall ((set ?X))
  (and (= (type.ftn:source (top ?X)) (constant-one ?X))
    (= (type.ftn:target (top ?X)) (power ?X))
    (= ((top ?X) iff:0) ?X)))

(iff:function binary-join)
(= (iff:source binary-join) set)
(= (iff:target binary-join) type.ftn:function)
(forall ((set ?X))
  (and (= (type.ftn:source (binary-join ?X)) (power-pair ?X))
    (= (type.ftn:target (binary-join ?X)) (power ?X))
    (forall (((power ?X) ?Y) ((power ?X) ?Z))
      (= ((binary-join ?X) [?Y ?Z]) (binary-union [?Y ?Z])))))

(iff:function binary-meet)
(= (iff:source binary-meet) set)
(= (iff:target binary-meet) type.ftn:function)
(forall ((set ?X))
  (and (= (type.ftn:source (binary-meet ?X)) (power-pair ?X))
    (= (type.ftn:target (binary-meet ?X)) (power ?X))
    (forall (((power ?X) ?Y) ((power ?X) ?Z))
      (= ((binary-meet ?X) [?Y ?Z]) (binary-intersection [?Y ?Z])))))

(iff:function minus)
(= (iff:source minus) set)
(= (iff:target minus) type.ftn:function)
(forall ((set ?X))
  (and (= (type.ftn:source (minus ?X)) (power-pair ?X))
    (= (type.ftn:target (minus ?X)) (power ?X))
    (forall (((power ?X) ?Y) ((power ?X) ?Z))
      (and (= ((minus ?X) [?Y ?Z]) (difference [?Y ?Z]))
        (= ((minus ?X) [?Y ?Z])
          (difference [?Y ?Z]))))))

```

```

((binary-meet ?X) [?Y ((complement ?X) ?Z)]))))))

(iff:function complement)
(= (iff:source complement) set)
(= (iff:target complement) type.ftn:function)
(forall ((set ?X))
  (and (= (type.ftn:source (complement ?X)) (power ?X))
        (= (type.ftn:target (complement ?X)) (power ?X))
        (forall ((power ?X) ?Y)
          (and (= ((complement ?X) ?Y) (difference [?X ?Y]))
                (= ((complement ?X) ?Y) ((minus ?X) [(top ?X) ?Y]))))))))

(iff:function implication)
(= (iff:source implication) set)
(= (iff:target implication) type.ftn:function)
(forall ((set ?X))
  (and (= (type.ftn:source (implication ?X)) (power-pair ?X))
        (= (type.ftn:target (implication ?X)) (power ?X))
        (forall ((power ?X) ?Y) ((power ?X) ?Z)
          (and (= ((implication ?X) [?Y ?Z])
                  (difference [?X (difference [?Y ?Z])]))
                (= ((implication ?X) [?Y ?Z])
                    ((complement ?X) ((difference ?X) [?Y ?Z])))
                (= ((implication ?X) [?Y ?Z])
                    (binary-join ?X [(complement ?X) ?Y] ?Z))))))

```

For any type set X , the *singleton* type function $\{-\}_X : X \rightarrow \wp X$ is defined as

$$\{x\}_X = \{x\}$$

for any element $x \in \wp X$. The singleton operation $\{-\}_X$ is an injection.

```

(iff:function singleton)
(= (iff:source singleton) set)
(= (iff:target singleton) type.ftn:function)
(forall ((set ?X))
  (and (= (type.ftn:source (singleton ?X)) ?X)
        (= (type.ftn:target (singleton ?X)) (power ?X))
        (forall ((?X ?x) (?X ?y))
          (<=> ((singleton ?X) ?x) ?y) (= ?y ?x)))
        (type.ftn:injection (singleton ?X))))

```

For any type set X , the (bounded) *union* (or *join*) type function $\cup_X : \wp \wp X \rightarrow \wp X$ is defined as

$$\cup_X(Z) = \{x \in X \mid \exists Y \in Z \ x \in Y\}$$

for any family of subsets $Z \in \wp \wp X$. The union operation \cup_X is a surjection, since for any $Y \in \wp X$ we have $\{Y\} \in \wp \wp X$ and $\cup_X(\{Y\}) = Y$; that is, $\exists_{\{-\}_X} \cdot \cup_X = 1_{\wp X} : \wp X \rightarrow \wp \wp X \rightarrow \wp X$.

```

(iff:function union) (iff:function join) (= join union)
(= (iff:source union) set)
(= (iff:target union) type.ftn:function)
(forall ((set ?X))
  (and (= (type.ftn:source (union ?X)) (power (power ?X)))
        (= (type.ftn:target (union ?X)) (power ?X)))

```

```

(forall ((power (power ?X)) ?Z) (?X ?x))
  (<=> ((union ?X) ?Z) ?x)
    (exists ((?Z ?Y) (?Y ?x))))
(type.ftn:surjection (union ?X)))

(forall ((set ?X) ((power ?X) ?Y))
  (= (type.ftn:composition [(exists (singleton ?X)) (union ?X)])
    (type.ftn:identity (power ?X))))

```

The triple $\langle \wp, \cup, \{-}\rangle$ forms a(n implicit) monad. This means that $\wp : \text{Set} \rightarrow \text{Set}$ is a(n implicit) functor, $\cup : \wp \circ \wp \Rightarrow \wp$ and $\{-\} : 1_{\text{Set}} \Rightarrow \wp$ are (implicit) natural transformations, and these satisfy the associative law and two unit laws

$$\begin{aligned}
\cup_{\wp(X)} \cdot \cup_X &= \wp(\cup_X) \cdot \cup_X \\
\wp(\{-\}_X) \cdot \cup_X &= 1_{\wp(X)} \\
\{-\}_{\wp(X)} \cdot \cup_X &= 1_{\wp(X)}
\end{aligned}$$

for all sets $X \in \text{set}$.

```

(forall ((set ?X))
  (and (= (type.ftn:composition [(union (power ?X)) (union ?X)])
    (type.ftn:composition [(type.ftn:power (union ?X)) (union ?X)]))
    (= (type.ftn:composition [(type.ftn:power (singleton ?X)) (union ?X)])
    (type.ftn:identity (power ?X)))
    (= (type.ftn:composition [(singleton (power ?X)) (union ?X)])
    (type.ftn:identity (power ?X))))))

```

For any type set X , the *intersection* (or *meet*) type function $\cap_X : \wp\wp X \rightarrow \wp X$ is defined as

$$\cap_X(Z) = \{x \in X \mid \forall Y \in Z \ x \in Y\}$$

for any family of subsets $Z \in \wp\wp X$. The intersection operation \cap_X is a surjection, since for any $Y \in \wp X$ we have $\{Y\} \in \wp\wp X$ and $\cap_X(\{Y\}) = Y$; that is, $\exists_{\{-\}_X} \cdot \cap_X = 1_{\wp X} : \wp X \rightarrow \wp\wp X \rightarrow \wp X$.

```

(iff:function intersection) (iff:function meet) (= meet intersection)
(= (iff:source intersection) set)
(= (iff:target intersection) type.ftn:function)
(forall ((set ?X))
  (and (= (type.ftn:source (intersection ?X)) (power (power ?X)))
    (= (type.ftn:target (intersection ?X)) (power ?X))
    (forall ((power (power ?X)) ?Z) (?X ?x))
      (<=> (((intersection ?X) ?Z) ?x)
        (forall ((?Z ?Y) (?Y ?x))))
    (type.ftn:surjection (intersection ?X))))

(forall ((set ?X) ((power ?X) ?Y))
  (= (type.ftn:composition [(exists (singleton ?X)) (intersection ?X)])
    (type.ftn:identity (power ?X))))

```

Let $f : X \rightarrow Y$ be any type function. There is an internal *existential* quantification, (direct) image or (*power*) type function $\exists_f : \wp X \rightarrow \wp Y$ defined by

$$y \in \exists_f A \equiv \begin{cases} \text{“there exists something } x \text{ in } X \\ \text{that satisfies } x \in A \text{ and} \\ \text{goes to } y \text{ under } f : f(x) = y\text{”} \end{cases}$$

or

$$\exists_f(A) = \{y \in Y \mid \exists_{x \in X}(x \in A \wedge f(x) = y)\}$$

for any source subset $A \subseteq X$. There is an internal *inverse image* type function

$$f^{-1} : \wp Y \rightarrow \wp X$$

defined by

$$\begin{aligned} f^{-1}(B) &= \{x \in X \mid \exists_{y \in B}(f(x) = y)\} \\ &= \{x \in X \mid f(x) \in B\} \end{aligned}$$

for any target subset $B \subseteq Y$. There is an internal universal quantification (*forall*) type function $\forall_f : \wp X \rightarrow \wp Y$ defined by

$$y \in \forall_f A \equiv \begin{cases} \text{“for all things } x \text{ in } X \\ \text{for which } f(x) = y, \\ x \in A \text{ holds”} \end{cases}$$

or

$$\forall_f(A) = \{y \in Y \mid \forall_{x \in X}(f(x) = y \Rightarrow A(x))\}$$

for any source subset $A \subseteq X$. The existential (inverse image, universal) function is monotonic. The existential quantification operation is functorial: the existential quantification of the identity $1_X : X \rightarrow X$ is the identity of the power, $\exists_{1_X} = 1_{\wp X} : \wp X \rightarrow \wp X$; and the existential quantification of the composition $f \cdot g : X \rightarrow Y \rightarrow Z$ is the composition of the existential quantifications, $\exists_{f \cdot g} = \exists_f \cdot \exists_g : \wp X \rightarrow \wp Z$. The inverse image (universal quantification) operation is also functorial. In the diagram

$$\begin{array}{ccc} & \xrightarrow{\exists_f} & \\ \wp X & \xleftarrow{f^{-1}} & \wp Y \\ & \xrightarrow{\forall_f} & \end{array}$$

we have the adjunctions $\exists_f \dashv f^{-1} \dashv \forall_f$; that is, the existential quantification is left adjoint to the inverse image and the inverse image is left adjoint to the universal quantification.

```
(iff:function exists) (iff:function power) (= power exists)
(= (iff:source exists) type.ftn:function)
(= (iff:target exists) type.ftn:function)
(forall ((type.ftn:function ?f))
  (and (= (type.ftn:source (exists ?f)) (power (type.ftn:source ?f)))
    (= (type.ftn:target (exists ?f)) (power (type.ftn:target ?f)))
    (forall (((power (type.ftn:source ?f)) ?A) ((type.ftn:target ?f) ?y))
      (<=> ((exists ?f) ?A) ?y)
      (exists (((type.ftn:source ?f) ?x)
        (and (?A ?x) (= (?f ?x) ?y)))))))

(iff:function inverse-image)
(= (iff:source inverse-image) type.ftn:function)
(= (iff:target inverse-image) type.ftn:function)
```

```

(forall ((type.ftn:function ?f))
  (and (= (type.ftn:source (inverse-image ?f)) (power (type.ftn:target ?f)))
    (= (type.ftn:target (inverse-image ?f)) (power (type.ftn:source ?f)))
    (forall ((power (type.ftn:target ?f)) ?B) ((type.ftn:source ?f) ?x)
      (<=> ((inverse-image ?f) ?B) ?x)
      (?B (?f ?x)))))

(iff:function forall)
(= (iff:source forall) type.ftn:function)
(= (iff:target forall) type.ftn:function)
(forall ((type.ftn:function ?f))
  (and (= (type.ftn:source (forall ?f)) (power (type.ftn:source ?f)))
    (= (type.ftn:target (forall ?f)) (power (type.ftn:target ?f)))
    (forall ((power (type.ftn:source ?f)) ?A) ((type.ftn:target ?f) ?y)
      (<=> ((forall ?f) ?A) ?y)
      (forall ((type.ftn:source ?f) ?x)
        (=> (= (?f ?x) ?y) (?A ?x)))))

(forall ((type.ftn:function ?f))
  (and (forall ((power (type.ftn:source ?f)) ?X0)
    ((power (type.ftn:source ?f)) ?X1)
    (subset-relation [?X0 ?X1]))
    (subset-relation [(exists ?f) ?X0] ((exists ?f) ?X1]))
  (forall ((power (type.ftn:target ?f)) ?Y0)
    ((power (type.ftn:target ?f)) ?Y1)
    (subset-relation [?Y0 ?Y1]))
    (subset-relation [(inverse-image ?f) ?Y0] ((inverse-image ?f) ?Y1]))
  (forall ((power (type.ftn:source ?f)) ?X0)
    ((power (type.ftn:source ?f)) ?X1)
    (subset-relation [?X0 ?X1]))
    (subset-relation [(forall ?f) ?X0] ((forall ?f) ?X1)))))

(forall ((type.ftn:function ?f))
  (and (forall ((power (type.ftn:source ?f)) ?X)
    ((power (type.ftn:target ?f)) ?Y)
    (<=> (subset-relation [(exists ?f) ?X] ?Y)
      (subset-relation [?X ((inverse-image ?f) ?Y)])))
    (forall ((power (type.ftn:source ?f)) ?X)
      ((power (type.ftn:target ?f)) ?Y)
      (<=> (subset-relation [(inverse-image ?f) ?Y] ?X)
        (subset-relation [?Y ((forall ?f) ?X)]))))

(forall ((set ?X))
  (= (exists (type.ftn:identity ?X)) (type.ftn:identity (power ?X))))

(forall ((type.ftn:function ?f) (type.ftn:function ?g) (composable-pair [?f ?g]))
  (= (exists (type.ftn:composition [?f ?g]))
    (type.ftn:composition [(exists ?f) (exists ?g)])))

```

1.3 Type Functions

Basics. There is an IFF set of all type *functions*. Any type function is itself an IFF function; that is, the IFF set of all type functions is an (implicit) subset of the set of all IFF functions.

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

Each type function has a unique *source* type set and a unique *target* type set. The source (target) map for type functions is an (implicit) restriction of the source (target) map for IFF functions. The (implicit) source-target pairing map for type functions is the (implicit) optimal restriction of the (implicit) source-target pairing map for IFF functions.

```
(iff:function source)
(= (iff:source source) function)
(= (iff:target source) type.set:set)
(forall ((function ?f))
  (= (source ?f) (iff:source ?f)))
```

```
(iff:function target)
(= (iff:source target) function)
(= (iff:target target) type.set:set)
(forall ((function ?f))
  (= (target ?f) (iff:target ?f)))
```

```
(forall ((iff:function ?f)
  (type.set:set (iff:source ?f))
  (type.set:set (iff:target ?f)))
  (function ?f))
```

There is a binary *restriction* relation \sqsubseteq between pairs of type functions that are linked by source and target inclusions. One (*smaller*) type function $f_1 : X_1 \rightarrow Y_1$ is a restriction of another (*larger*) type function $f_2 : X_2 \rightarrow Y_2$, $f_1 \sqsubseteq f_2$, when (1) the source (target) of f_1 is a subset of the source (target) of f_2 , $X_1 \subseteq X_2$ and $Y_1 \subseteq Y_2$, and (2) the functions agree (on source elements of f_1), $f_1(x_1) = f_2(x_1)$ for all elements $x_1 \in X_1$; that is, the functions commute with the source/target inclusions. This implies that the source type set of f_1 is a subset of the inverse image along f_2 of the target type set of f_1 : $X_1 \subseteq f_2^{-1}(Y_1)$. We name the components of a restriction relationship.

$$\begin{array}{ccc}
 X_1 & \xrightarrow{f_1} & Y_1 \\
 \downarrow & & \downarrow \\
 X_2 & \xrightarrow{f_2} & Y_2
 \end{array}$$

From one point of view, restriction can be viewed as a constraint on the larger type function — it says that the larger function maps the source type set of the smaller function into the target type set of the smaller function. From another point of view, restriction defines the smaller function; that is, if we assume that the definition of the larger function is given (in some other axiomatization), then asserting the restriction relationship effectively defines the smaller function. This is the point of view taken when we use (only) restriction to define a particular function at one metalevel in terms of the corresponding function at

the next higher metalevel. The restriction relation is a partial order (reflexive, antisymmetric and transitive), since the subset relation is a partial order.

```
(iff:set restriction-relation)
(forall ((restriction-relation ?f12)) (type.dgm.pr.mor:function-pair ?f12))
(forall ((function ?f1) (function ?f2))
  (<=> (restriction-relation [?f1 ?f2])
    (and (type.set:subordinate [(source ?f1) (source ?f2)])
      (type.set:subordinate [(target ?f1) (target ?f2)])
      (= (composition [?f1 (type.set:inclusion [(target ?f1) (target ?f2)])])
        (composition [(type.set:inclusion [(source ?f1) (source ?f2)]) ?f2])))))
(forall ((function ?f1) (function ?f2) (restriction-relation [?f1 ?f2]))
  (type.set:subordinate [(source ?f1) ((inverse-image ?f2) (target ?f1))]))

(iff:function smaller)
(= (iff:source smaller) restriction-relation)
(= (iff:target smaller) function)
(forall ((function ?f1) (function ?f2) (restriction-relation [?f1 ?f2]))
  (= (smaller [?f1 ?f2]) ?f1))

(iff:function larger)
(= (iff:source larger) restriction-relation)
(= (iff:target larger) function)
(forall ((function ?f1) (function ?f2) (restriction-relation [?f1 ?f2]))
  (= (larger [?f1 ?f2]) ?f2))

(forall ((function ?f))
  (restriction-relation [?f ?f]))
(forall ((function ?f1) (function ?f2))
  (=> (and (restriction-relation [?f1 ?f2]) (restriction-relation [?f2 ?f1]))
    (= ?f1 ?f2)))
(forall ((function ?f1) (function ?f2) (function ?f3))
  (=> (and (restriction-relation [?f1 ?f2]) (restriction-relation [?f2 ?f3]))
    (restriction-relation [?f1 ?f3])))
```

There is a binary *optimal restriction* relation $\dot{\sqsubseteq}$ between pairs of type functions. A restriction between two type functions f_1 and f_2 is optimal, $f_1 \dot{\sqsubseteq} f_2$, when the source type set of the smaller function is exactly the inverse image of the target type set of the smaller function along the larger function: $X_1 = f_2^{-1}(Y_1)$. Optimal restriction is a pullback notion. We name the components of an optimal restriction relationship.

$$\begin{array}{ccc} X_1 & \xrightarrow{f_1} & Y_1 \\ \downarrow & & \downarrow \\ X_2 & \xrightarrow{f_2} & Y_2 \end{array} \quad \lrcorner$$

The optimal restriction relation is also a partial order (reflexive, antisymmetric and transitive).

```
(iff:set optimal-restriction-relation)
(forall ((optimal-restriction-relation ?f12)) (restriction-relation ?f12))
(forall ((function ?f1) (function ?f2) (restriction-relation [?f1 ?f2]))
  (<=> (optimal-restriction-relation [?f1 ?f2])
    (= (source ?f1) ((inverse-image ?f2) (target ?f1)))))
```

```

(iff:function optimal-smaller)
(= (iff:source optimal-smaller) optimal-restriction-relation)
(= (iff:target optimal-smaller) function)
(forall ((function ?f1) (function ?f2) (optimal-restriction-relation [?f1 ?f2]))
  (= (optimal-smaller [?f1 ?f2]) ?f1))

(iff:function optimal-larger)
(= (iff:source optimal-larger) optimal-restriction-relation)
(= (iff:target optimal-larger) function)
(forall ((function ?f1) (function ?f2) (optimal-restriction-relation [?f1 ?f2]))
  (= (optimal-larger [?f1 ?f2]) ?f2))

(forall ((function ?f))
  (optimal-restriction-relation [?f ?f]))
(forall ((function ?f1) (function ?f2))
  (=> (and (optimal-restriction-relation [?f1 ?f2]) (optimal-restriction-relation [?f2 ?f1]))
    (= ?f1 ?f2)))
(forall ((function ?f1) (function ?f2) (function ?f3))
  (=> (and (optimal-restriction-relation [?f1 ?f2]) (optimal-restriction-relation [?f2 ?f3]))
    (optimal-restriction-relation [?f1 ?f3])))

```

Both restriction and optimal restriction are closed under composition and identities. If $f_1 : X_1 \rightarrow Y_1$ and $g_1 : Y_1 \rightarrow Z_1$ is a composable pair of type functions, $f_2 : X_2 \rightarrow Y_2$ and $g_2 : Y_2 \rightarrow Z_2$ is a composable pair of type functions, f_1 is a (the optimal-)restriction of f_2 and g_1 is a (the optimal-)restriction of g_2 , $f_1 \sqsubseteq (\overset{\square}{\subseteq}) f_2$ and $g_1 \sqsubseteq (\overset{\square}{\subseteq}) g_2$, then the composition $f_1 \cdot g_1 : X_1 \rightarrow Z_1$ is a (the optimal-)restriction of the composition $f_2 \cdot g_2 : X_2 \rightarrow Z_2$, $f_1 \cdot g_1 \sqsubseteq (\overset{\square}{\subseteq}) f_2 \cdot g_2$. If X_1 is a subset of X_2 , $X_1 \subseteq X_2$, then the identity $1_{X_1} : X_1 \rightarrow X_1$ is a (the optimal-)restriction of the identity $1_{X_2} : X_2 \rightarrow X_2$, $1_{X_1} \sqsubseteq (\overset{\square}{\subseteq}) 1_{X_2}$.

```

(forall ((function ?f1) (function ?g1) (composable-pair [?f1 ?g1])
  (function ?f2) (function ?g2) (composable-pair [?f2 ?g2])
  (restriction-relation [?f1 ?f2]) (restriction-relation [?g1 ?g2]))
  (restriction-relation [(composition [?f1 ?g1]) (composition [?f2 ?g2])]))

(forall ((type.set:set ?X1) (type.set:set ?X2) (type.set:subordinate [?X1 ?X2]))
  (restriction-relation [(identity ?X1) (identity ?X2)]))

(forall ((function ?f1) (function ?g1) (composable-pair [?f1 ?g1])
  (function ?f2) (function ?g2) (composable-pair [?f2 ?g2])
  (optimal-restriction-relation [?f1 ?f2]) (optimal-restriction-relation [?g1 ?g2]))
  (optimal-restriction-relation [(composition [?f1 ?g1]) (composition [?f2 ?g2])]))

(forall ((type.set:set ?X1) (type.set:set ?X2) (type.set:subordinate [?X1 ?X2]))
  (optimal-restriction-relation [(identity ?X1) (identity ?X2)]))

```

Category theory represents elements-in-sets by morphisms⁵. A type *element* x in a set $X \in \mathbf{set}$ is a type function $1 \xrightarrow{x} X$. We use the notation $\sigma(x) = X$ or the notation $x \in X$ to denote this. There is an IFF set *elmt* of all type elements, which is a(n implicit) subset of the IFF set of all functions, $\mathbf{elmt} \subseteq \mathbf{ftn}$. There is a *set* function $\sigma : \mathbf{elmt} \rightarrow \mathbf{set}$. The set of an element is its target, $\sigma(x) = \partial_1(x)$

⁵Note that these elements are not separate individuals, but have a set attached. This provides a kind of context for the element. The representation of elements as separate individuals is not needed in the natural part of the IFF. The categorical representation of elements within a context is exactly what is needed.

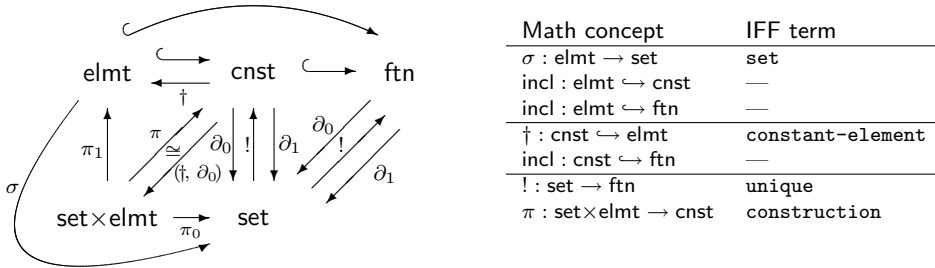


Figure 1: Basic Function Kinds

(outer part of Figure 1). X as a set-theoretic “bag of dots” corresponds to the fiber of the inclusion function over X .

```
(iff:set element)
(forall ((element ?x)) (function ?x))
(forall ((function ?x))
  (<=> (element ?x)
    (= (source ?x) type.set:one)))
```

```
(iff:function set)
(= (iff:source set) element)
(= (iff:target set) type.set:set)
(forall ((element ?x))
  (= (set ?x) (target ?x)))
```

The sets two and three have some useful elements mapping into their contents:

```
(element two0)
(= (set two0) type.set:two)
(= (two0 iff:0) iff:0)

(element two1)
(= (set two1) type.set:two)
(= (two0 iff:0) iff:1)

(element three0)
(= (set three0) type.set:three)
(= (three0 iff:0) iff:0)

(element three1)
(= (set three1) type.set:three)
(= (three1 iff:0) iff:1)

(element three2)
(= (set three2) type.set:three)
(= (three2 iff:0) iff:2)
```

The statement that “the function $c : Y \rightarrow X$ is constant” means that there exists an element of $x \in X$ such that c is the composite $c = !_Y \cdot x : Y \rightarrow 1 \rightarrow X$. We use the notation $\dagger(c) = x$ to denote the element x associated with a constant function c . Note that there is no constraint between the set Y and the element $x \in X$; they are completely independent. Conversely, given any pair $(Y, x \in X)$, consisting of a type set $Y \in \text{set}$ and a type element $x \in \text{elmt}$, we can construct a constant function $!_Y \cdot x : Y \rightarrow 1 \rightarrow X$. As two special cases, any type element

$x \in \text{elmt}$ is a constant function $x : 1 \rightarrow X$, and any type set $Y \in \text{set}$ has an associated constant function $!_Y : Y \rightarrow 1$. There is an IFF set cnst of all *constant* type functions, which is a(n implicit) subset of the IFF set of type functions, $\text{cnst} \subseteq \text{ftn}$. The IFF set of type elements is a(n implicit) subset of the IFF set of constant type functions, $\text{elmt} \subseteq \text{cnst}$. There is an element function $\dagger : \text{cnst} \rightarrow \text{elmt}$. The IFF set of constant type functions is (implicitly) the binary product of the IFF set of type sets and the IFF set of type elements, $\text{cnst} \cong \text{set} \times \text{elmt}$. This isomorphism is mediated by the pairing $(\dagger, \partial_0) : \text{cnst} \rightarrow \text{set} \times \text{elmt}$ and the *construction* function $\pi : \text{set} \times \text{elmt} \rightarrow \text{cnst}$.

```
(iff:set constant-function)
(forall ((constant-function ?c)) (function ?c))
(forall ((function ?c))
  (<=> (constant-function ?c)
    (exists ((element ?x))
      (= ?c (composition [(type.set:unique (source ?c)) ?x])))))

(iff:function constant-element)
(= (iff:source constant-element) constant-function)
(= (iff:target constant-element) element)
(forall ((constant-function ?c))
  (and (= (set (constant-element ?c)) (target ?c))
    (= ?c (composition [(type.set:unique (source ?c)) (constant-element ?c)]))))

(iff:function construction)
(forall ((iff:source construction) ?Yx))
  (exists ((type.set:set ?Y) (element ?x))
    (= ?Yx [?Y ?x]))
(forall ((type.set:set ?Y) (element ?x))
  ((iff:source construction) [?Y ?x]))
(= (iff:target construction) constant-function)
(forall ((type.set:set ?Y) (element ?x))
  (and (= (construction [?Y ?x]) (composition [(type.set:unique ?Y) ?x]))
    (= (constant-element (construction [?Y ?x])) ?x)))

(forall ((type.set:set ?Y)) (constant-function (type.set:unique ?Y)))
(forall ((element ?x)) (constant-function ?x))
```

Category Theory. A *pair* of type functions is *composable* when the target of the first is equal to the source of the second. We name the projection *factors* of composable pairs.

```
(iff:set composable-pair)
(forall ((composable-pair ?fg)) (type.dgm.pr.mor:function-pair ?fg))
(forall ((function ?f) (function ?g))
  (<=> (composable-pair [?f ?g])
    (= (target ?f) (source ?g))))

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

(iff:function factor1)
(= (iff:source factor1) composable-pair)
(= (iff:target factor1) function)
```

```
(forall ((function ?f) (function ?g) (composable-pair [?f ?g]))
  (= (factor1 [?f ?g]) ?g))
```

The *composition* of two composable type functions $f : X \rightarrow Y$ and $g : Y \rightarrow Z$ is a type function $f \cdot g : X \rightarrow Z$. The source of the composite is the source of the first component factor, and the target of the composite is the target of the second component factor.

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

Composition is associative. Any three composable type functions $f : X \rightarrow Y$, $g : Y \rightarrow Z$ and $h : Z \rightarrow W$ satisfy the associative law $f \cdot (g \cdot h) = (f \cdot g) \cdot h$.

```
(forall ((function ?f) (function ?g) (function ?h)
  (composable-pair [?f ?g]) (composable-pair [?g ?h]))
  (= (composition [?f (composition [?g ?h])]
    (composition [(composition [?f ?g]) ?h])))
```

The composition map $\text{ftn} \times_{\text{set}} \text{ftn} \rightarrow \text{ftn}$ is surjective (see identity below).

```
(forall ((function ?h)
  (function ?f) (function ?g) (composable-pair [?f ?g]))
  (= (composition [?f ?g]) ?h))
```

For every type set X , there is a unique associated *identity* type function $1_X : X \rightarrow X$.

```
(iff:function identity)
(= (iff:source identity) type.set:set)
(= (iff:target identity) function)
(forall ((type.set:set ?X))
  (and (= (source (identity ?X)) ?X)
    (= (target (identity ?X)) ?X)))
```

Identity satisfies two unit laws with respect to composition: composition with the identity of the source (target) of a function $f : X \rightarrow Y$ returns that function: $1_X \cdot f = f = f \cdot 1_Y$.

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

For any set X , the identity function $1_X : X \rightarrow X$ is a bijection.

```
(forall ((type.set:set ?X))
  (type.ftn:bijection (identity ?X)))
```

The identity map is injective $\text{set} \xrightarrow{1} \text{ftn}$. Hence, sets can be regarded as special functions that satisfy the unit laws.

```
(forall ((type.set:set ?X0) (type.set:set ?X1))
  (= (identity ?X0) (identity ?X1)))
(= ?X0 ?X1)
```

Application gives the definition of any function. For element x and function f , the expression ' $f(x)$ ' is not defined. We use application to give this meaning. A pair (x, f) consisting of a type element x and a type function f is *applicable* when the set of the element is equal to the source of the function, $\sigma(x) = \partial_0(f)$. The IFF set of all applicable pairs is a(n implicit) subset of the IFF set of all composable pairs. We name the projection *factors* of applicable pairs.

```
(iff:set applicable-pair)
(forall ((applicable-pair ?xf)
         (composable-pair ?xf))
 (forall ((element ?x) (function ?f))
  (<=> (applicable-pair [?x ?f])
       (= (set ?x) (source ?f))))
```

```
(iff:function element-factor)
(= (iff:source element-factor) applicable-pair)
(= (iff:target element-factor) element)
(forall ((element ?x) (function ?f) (applicable-pair [?x ?f]))
 (= (element-factor [?x ?f]) ?x))
```

```
(iff:function function-factor)
(= (iff:source function-factor) applicable-pair)
(= (iff:target function-factor) function)
(forall ((element ?x) (function ?f) (applicable-pair [?x ?f]))
 (= (function-factor [?x ?f]) ?f))
```

The type set of all applicable pairs is a(n implicit) subset of the type set of all composable pairs, since $\sigma(x) = \partial_0(f)$.

```
(forall ((element ?x) (function ?f) (applicable-pair [?x ?f]))
 (composable-pair [?x ?f]))
```

The *application* of an applicable pair $x \in X$ and $f : X \rightarrow Y$ is an element $(x \wr f) \in Y$. The set of the result(ing element) is the target of the function. The embedding of a result is the composition of the input element: $x \wr f = x \cdot f$ for any applicable pair $x \in X$ and $f : X \rightarrow Y$; that is, application is a restriction of composition.

```
(iff:function application)
(= (iff:source application) applicable-pair)
(= (iff:target application) element)
(forall ((element ?x) (function ?f) (applicable-pair [?x ?f]))
 (= (set (application [?x ?f]) (target ?f)))
 (forall ((element ?x) (function ?f) (applicable-pair [?x ?f]))
  (= (application [?x ?f]) (composition [?x ?f]))))
```

There is a mixed associative law for application. Using the associative law for composition, $x \wr (f \cdot g) = (x \wr f) \wr g$ for any composable pair $f : X \rightarrow Y$ and $g : Y \rightarrow Z$ and any element $x \in X$. This corresponds to the usual pointwise definition of composition, $(f \cdot g)(x) = g(f(x))$. Hence, the associative law for composition implies the definition of application of composition.

```
(forall ((element ?x) (function ?f) (function ?g)
         (applicable-pair [?x ?f]) (composable-pair ?f ?g))
 (= (application [?x (composition [?f ?g])]
              (application [(application [?x ?f]) ?g])))
```

There is a unit law for application. Using the unit laws for composition, $x \wr 1_X = x$ for any set X and any element $x \in X$. This corresponds to the usual pointwise definition of identity, $1_X(x) = x$. Hence, the unit laws for composition imply the definition of application of identity.

```
(forall ((element ?x))
  (= (application [?x (identity (set ?x))] ?x))
```

Factorization. There are special types of functions. A type function $f : X \rightarrow Y$ is an *injection* (monomorphism) when any image value is from a unique source element; or more categorically, when for any parallel pair of type functions (generalized elements) $x_0, x_1 : W \rightarrow X$ the equality $x_0 \cdot f = x_1 \cdot f$ implies $x_0 = x_1$ (f is right-cancellable). A type function $f : X \rightarrow Y$ is a *surjection* (epimorphism) when any target value is an image; or more categorically, when for any parallel pair of type functions $y_0, y_1 : Y \rightarrow Z$ the equality $f \cdot y_0 = f \cdot y_1$ implies $y_0 = y_1$ (f is left-cancellable). A type function $f : X \rightarrow Y$ is a *bijection* (isomorphism) when there is a (necessarily unique *inverse*) function in the opposite direction $\hat{f} : Y \rightarrow X$ with $f \cdot \hat{f} = 1_X$ and $\hat{f} \cdot f = 1_Y$.

```
(iff:set injection)
(forall ((injection ?f)) (function ?f))
(forall ((function ?f))
  (<=> (injection ?f)
    (forall ((function ?x0) (function ?x1)
      (composable-pair [?x0 ?f]) (composable-pair [?x1 ?f])
      (= (composition [?x0 ?f]) (composition [?x1 ?f])))
    (= ?x0 ?x1))))

(iff:set surjection)
(forall ((surjection ?f)) (function ?f))
(forall ((function ?f))
  (<=> (surjection ?f)
    (forall ((function ?y0) (function ?y1)
      (composable-pair [?f ?y0]) (composable-pair [?f ?y1])
      (= (composition [?f ?y0]) (composition [?f ?y1])))
    (= ?y0 ?y1))))

(iff:set bijection)
(forall ((bijection ?f)) (function ?f))
(forall ((function ?f))
  (<=> (bijection ?f)
    (exists ((function ?fh)) (composable-pair [?f ?fh]) (composable-pair [?fh ?f]))
    (and (= (composition [?f ?fh]) (identity (source ?f)))
      (= (composition [?fh ?f]) (identity (target ?f))))))

(iff:function inverse)
(= (iff:source inverse) bijection)
(= (iff:target inverse) bijection)
(forall ((bijection ?f))
  (and (= (source (inverse ?f)) (target ?f))
    (= (target (inverse ?f)) (source ?f))
    (= (composition [?f (inverse ?f)]) (identity (source ?f)))
    (= (composition [(inverse ?f) ?f]) (identity (target ?f)))))

(forall ((bijection ?f))
  (= (inverse (inverse ?f)) ?f))
```

A function is a bijection when it is both an injection and a surjection.

```
(forall ((function ?f))
  (<=> (bijection ?f)
    (and (injection ?f) (surjection ?f))))
```

Injections, surjections and bijections are closed under composition.

```
(forall ((injection ?f) (injection ?g) (composable-pair [?f ?g]))
  (injection (composition [?f ?g])))
(forall ((surjection ?f) (surjection ?g) (composable-pair [?f ?g]))
  (surjection (composition [?f ?g])))
(forall ((bijection ?f) (bijection ?g) (composable-pair [?f ?g]))
  (bijection (composition [?f ?g])))
```

For any type function $f : X \rightarrow Y$, there is a *range* type set $\rho_f = \text{rng}(f) \subseteq Y$, defined by $\rho(f) = \{y \in Y \mid \exists x \in X f(x) = y\}$.

```
(iff:function range)
(= (iff:source range) function)
(= (iff:target range) type.set:set)
(forall ((function ?f))
  (and (subset-relation [(range ?f) (target ?f)])
    (forall ((target ?f) ?y)
      (<=> ((range ?f) ?y)
        (exists ((source ?f) ?x)
          (= ?y (?f ?x)))))))
```

For any type function $f : X \rightarrow Y$, there is an *injective factor* $\iota_f = \text{inj}_f : \rho(f) \rightarrow Y$, which is the inclusion of the range of f into the target of f .

```
(iff:function injective-factor)
(= (iff:source injective-factor) function)
(= (iff:target injective-factor) function)
(forall ((function ?f))
  (and (= (source (injective-factor ?f)) (range ?f))
    (= (target (injective-factor ?f)) (target ?f))
    (type.ftn:injection (injective-factor ?f))
    (= (injective-factor ?f) (type.set:inclusion [(range ?f) (target ?f)]))))
```

For any type function $f : X \rightarrow Y$, there is a *surjective factor* $\sigma_f = \text{surj}_f : X \rightarrow \rho(f)$ with the same action as f (it is the target restriction of f to its range).

```
(iff:function surjective-factor)
(= (iff:source surjective-factor) function)
(= (iff:target surjective-factor) function)
(forall ((function ?f))
  (and (= (source (surjective-factor ?f)) (source ?f))
    (= (target (surjective-factor ?f)) (range ?f))
    (surjection (surjective-factor ?f))
    (restriction-relation [(surjective-factor ?f) ?f])))
```

The function f is the composition of its surjective factor and injective factor: $f = \sigma_f \cdot \iota_f : X \rightarrow \rho(f) \rightarrow Y$.

```
(forall ((function ?f))
  (= ?f (composition [(surjective-factor ?f) (injective-factor ?f)])))
```

The (surjective-factor, injective-factor) pair forms a surjection-injection “factorization system” for type sets and type functions; that is, if a type function $f : X \rightarrow Y$ is the composition of a surjection with an injection, $f = e \cdot m : X \rightarrow Z \rightarrow Y$, then there is a (unique) “diagonal” bijection $d : \rho(f) \rightarrow Z$, such that $\sigma_f \cdot d = e$ and $d \cdot m = \iota_f$.

$$\begin{array}{ccc}
X & \xrightarrow{\sigma_f} & \rho(f) \\
e \downarrow & \swarrow d & \downarrow \iota_f \\
Z & \xrightarrow{m} & Y
\end{array}$$

```

(forall ((function ?f) (surjection ?e) (injection ?m) (= ?f (composition [?e ?m])))
  (and (exists ((bijection ?d) (= (source ?d) (range ?f)) (= (target ?d) (target ?e)))
    (and (= (composition [(surjective-factor ?f) ?d]) ?e)
      (= (composition [?d ?m]) (injective-factor ?f))))
    (forall ((bijection ?d1) (bijection ?d2)
      (= (source ?d1) (range ?f)) (= (target ?d1) (target ?e))
      (= (source ?d2) (range ?f)) (= (target ?d2) (target ?e))
      (= (composition [(surjective-factor ?f) ?d1]) ?e)
      (= (composition [(surjective-factor ?f) ?d2]) ?e)
      (= (composition [?d1 ?m]) (injective-factor ?f))
      (= (composition [?d2 ?m]) (injective-factor ?f))
      (= ?d1 ?d2))))

```

Let $f : X \rightarrow Y$ be a type function. The *kernel* of f is the equivalence relation $\ker_f = \equiv_f$ on X defined by $x_1 \equiv_f x_2$ iff $f(x_1) = f(x_2)$ for all source pairs $x_1, x_2 \in X$. The *coimage* of f is the quotient of the kernel $\text{coim}_f = X/\equiv_f = \{[x]_{\equiv_f} \mid x \in X\}$, where $[x]_f = \{x' \in X \mid f(x') = f(x)\}$ is the equivalence class of x with respect to the kernel of f , and the *coequalizer*⁶ of f is the canon of the kernel $\text{coeq}_f = [-]_f = [-]_{\equiv_f} : X \rightarrow \text{coim}_f$.

```

(iff:function kernel)
(= (iff:source kernel) function)
(= (iff:target kernel) type.rel:equivalence-relation)
(forall ((function ?f))
  (and (= (type.rel:set (kernel ?f)) (source ?f))
    (forall (((source ?f) ?x1) ((source ?f) ?x2))
      (<=> ((kernel ?f) ?x1 ?x2)
        (= (?f ?x1) (?f ?x2))))))

```

```

(iff:function coimage)
(= (iff:source coimage) function)
(= (iff:target coimage) type.set:set)
(forall ((function ?f))
  (= (coimage ?f) (type.rel:quotient (kernel ?f))))

```

```

(iff:function coequalizer)
(= (iff:source coequalizer) function)
(= (iff:target coequalizer) function)
(forall ((function ?f))
  (and (= (source (coequalizer ?f)) (source ?f))
    (= (target (coequalizer ?f)) (coimage ?f))
    (surjection (coequalizer ?f))
    (= (coequalizer ?f) (type.rel:canon (kernel ?f)))))

```

Any function $f : X \rightarrow Y$ respects its kernel $\pi_0^{\equiv_f} \cdot f = \pi_1^{\equiv_f} \cdot f$, and hence factors through its coimage $f = [-]_f \cdot \lambda_f$ for some (injective) function $\text{coapply}_f = \lambda_f : \text{coim}_f \rightarrow Y$. This factor, called the *coapplication* of f , is defined by $\lambda_f([x]_{\equiv_f}) = f(x)$.

⁶So named because it is the coequalizer of the kernel projection pair.

$$\begin{array}{ccc}
\text{ext}(\equiv_f) & \xrightarrow[\pi_1]{\pi_0} & X & \xrightarrow{f} & Y \\
& & \searrow [-]_f & & \nearrow \wr_f \\
& & & & X/\equiv_f
\end{array}$$

```

(iff:function coapplication)
(= (iff:source coapplication) function)
(= (iff:target coapplication) function)
(forall ((function ?f))
  (and (= (source (coapplication ?f)) (coimage ?f))
        (= (target (coapplication ?f)) (target ?f))
        (injection (coapplication ?f))
        (= ?f (composition [(coequalizer ?f) (coapplication ?f)]))))

```

The (coequalizer, coapplication) pair forms a surjection-injection “factorization system” for type sets and type functions; that is, if a type function $f : X \rightarrow Y$ is the composition of a surjection with an injection, $f = e \cdot m : X \rightarrow Z \rightarrow Y$, then there is a (unique) “diagonal” bijection $d : \text{coim}(f) \rightarrow Z$, such that $[-]_f \cdot d = e$ and $d \cdot m = \wr_f$.

$$\begin{array}{ccc}
X & \xrightarrow{[-]_f} & X/\equiv_f \\
e \downarrow & \swarrow d & \downarrow \wr_f \\
Z & \xrightarrow{m} & Y
\end{array}$$

This implies that the coimage is naturally isomorphic to the range (image), $\text{coim}_f \cong \text{rng}_f$; specifically, for any source element of $x \in X$, the equivalence class $[x]_{\equiv_f} \in \text{coim}_f$ corresponds to the image element $f(x) \in \text{rng}_f$.

Conversion. Any function $X \xrightarrow{f} Y$ has an associated image *predicate*, whose genus is the target of the function, whose differentia is the range of the function, and whose injection is the injective-factor of the function.

```

(iff:function predicate)
(= (iff:source predicate) function)
(= (iff:target predicate) type.pred:predicate)
(forall ((function ?f))
  (and (= (type.pred:genus (predicate ?f)) (target ?f))
        (= (type.pred:differentia (predicate ?f)) (range ?f))
        (= (type.pred:function (predicate ?f)) (injective-factor ?f))))

```

For any type function $x : Y \xrightarrow{x} X$, there is a *unit* function 2-cell $\eta_x : x \Rightarrow \text{ftn}(\text{pred}(x))$ whose source is x , whose target $\text{ftn}(\text{pred}(x))$ is the function (injection) of the predicate of x , and whose function $\sigma_x : X \rightarrow \varepsilon(x)$ is the surjective factor of x . (Figure 2).

```

(iff:function unit)
(= (iff:source unit) function)
(= (iff:target unit) type.ftn.mor:2-cell)
(forall ((function ?x))
  (and (= (type.ftn.mor:source (unit ?r)) ?x)
        (= (type.ftn.mor:target (unit ?r)) (type.pred:function (predicate ?x)))
        (= (type.ftn.mor:function (unit ?r)) (surjective-factor ?x))))

```

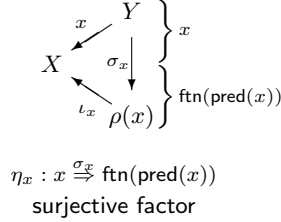


Figure 2: Unit Function 2-Cell

There is a function to *span* injection that embeds a type function $f : X \rightarrow Y$ as a type span $\text{spn}(f) = \langle X \xleftarrow{1_X} X \xrightarrow{f} Y \rangle$.

```
(iff:function span)
(= (iff:source span) function)
(= (iff:target span) type.spn:span)
(forall ((function ?f))
  (and (= (type.spn:function0 (span ?f)) (identity (source ?f)))
        (= (type.spn:function1 (span ?f)) ?f)
        (= (type.spn:vertex (span ?f)) (source ?f))))
```

There is a function to *relation* map that embeds a type function $f : X \rightarrow Y$ as a total functional relation $\text{rel}(f) = \hat{f} : X \rightarrow Y$, where $\text{ext}(\text{rel}(f)) = \{(x, y) \mid x \in X, y \in Y, f(x) = y\}$. The domain (codomain) of the relation is the source (target) of the function. The domain projection is a bijection, and post-composition with the original function gives the codomain projection. The relation of a function is the relation of the span of the function. The relation map $\text{ftn} \rightarrow \text{rel}$ is injective.

```
(iff:function relation)
(= (iff:source relation) function)
(= (iff:target relation) type.rel:relation)
(forall ((function ?f))
  (and (= (type.rel:set0 (relation ?f)) (source ?f))
        (= (type.rel:set1 (relation ?f)) (target ?f))
        (type.set:isomorphic-relation [(type.rel:extent (relation ?f)) (source ?f)])
        (bijection (type.rel:projection0 (relation ?f)))
        (= (composition [(type.rel:projection0 (relation ?f)) ?f])
            (type.rel:projection1 (relation ?f)))
        (= (relation ?f) (type.spn:relation (span ?f)))
        (forall (((source ?f) ?x) ((target ?f) ?y))
          (<=> ((relation ?f) ?x ?y)
                (= (?f ?x) ?y))))))
(forall ((function ?f1) (function ?f2) (= (relation ?f1) (relation ?f2)))
  (= ?f1 ?f2))
```

For any type function $f : X \rightarrow Y$, there is a *fiber* type function $\text{fbr}(f) : Y \rightarrow \wp X$, defined as

$$\text{fbr}(f)(y) = \{x \in X \mid f(x) = y\}$$

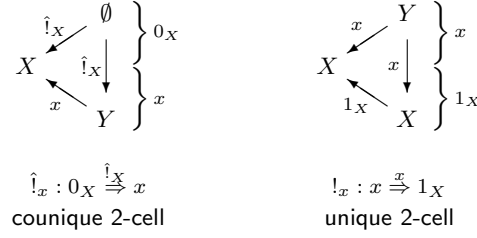


Figure 3: Counique and Unique Function 2-Cells

for any target element $y \in Y$. The fiber function can be defined in terms of target singleton and inverse image, $\text{fbr}(f) = \{-\}_Y \cdot f^{-1} : Y \rightarrow \wp Y \rightarrow \wp X$. The fiber function of $f : X \rightarrow Y$ is the 10-fiber of the relation $\hat{f} : X \rightarrow Y$.

```

(iff:function fiber)
(= (iff:source fiber) function)
(= (iff:target fiber) function)
(forall ((function ?f))
  (and (= (source (fiber ?f)) (target ?f))
        (= (target (fiber ?f)) (type.set:power (source ?f)))
        (= (fiber ?f) (type.rel:fiber01 (relation ?f)))
        (forall (((target ?f) ?y) ((source ?f) ?x))
          (<=> (((fiber ?f) ?y) ?x)
                (= (?f ?x) ?y))))))

(forall ((function ?f))
  (= (fiber ?f)
      (composition [(type.set:singleton (target ?f)) (type.set:inverse-image ?f)])))
  
```

Instances. For any set X , the empty set and counique function form an *initial* (predicative) function $0_X = \emptyset \xrightarrow{\hat{!}_X} X$, and the identity function forms a *terminal* (predicative) function $1_X = X \xrightarrow{!_X} X$.

```

(iff:function initial)
(= (iff:source initial) type.set:set)
(= (iff:target initial) function)
(forall ((type.set:set ?X))
  (and (= (source (initial ?X)) type.set:zero)
        (= (target (initial ?X)) ?X)
        (= (initial ?X) (type.set:counique ?X))))

(iff:function terminal)
(= (iff:source terminal) type.set:set)
(= (iff:target terminal) function)
(forall ((type.set:set ?X))
  (and (= (source (terminal ?X)) ?X)
        (= (target (terminal ?X)) ?X)
        (= (terminal ?X) (identity ?X))))
  
```

For any type function $x = (Y \xrightarrow{x} X)$, there is a *counique* type function 2-cell $\hat{!}_x : 0_X \rightrightarrows x$ and a *unique* type function 2-cell $!_x : x \rightrightarrows 1_X$ (Figure 3). These are the unique function 2-cells between their respective sources and targets.

```

(iff:function counique)
(= (iff:source counique) function)
(= (iff:target counique) type.ftn.mor:2-cell)
(forall ((function ?x))
  (and (= (type.ftn.mor:source (counique ?x)) (initial (target ?x)))
        (= (type.ftn.mor:target (counique ?x)) ?x)
        (= (type.ftn.mor:function (counique ?x)) (type.set:counique (target ?x))))
  (forall ((type.ftn.mor:2-cell ?a)
            (= (type.ftn.mor:source ?a) (initial (target ?x)))
            (= (type.ftn.mor:target ?a) ?x))
    (= ?a (counique ?x))))

(iff:function unique)
(= (iff:source unique) function)
(= (iff:target unique) type.ftn.mor:2-cell)
(forall ((function ?x))
  (and (= (type.ftn.mor:source (unique ?x)) ?x)
        (= (type.ftn.mor:target (unique ?x)) (terminal (target ?x)))
        (= (type.ftn.mor:function (unique ?x)) ?x)
        (forall ((type.ftn.mor:2-cell ?a)
                  (= (type.ftn.mor:source ?a) ?x)
                  (= (type.ftn.mor:target ?a) (terminal (target ?x)))
                  (= ?a (unique ?x))))))

```

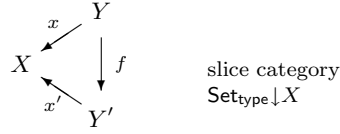


Figure 4: Function 2-Cell

1.3.1 Function Morphisms

Basics. A type function *morphism* is a morphism in the comma category⁷

$$\mathbf{Set}_{\text{type}} \xleftarrow{\pi_0} (1 \downarrow 1) \xrightarrow{\pi_1} \mathbf{Set}_{\text{type}}$$

over the functorial ospan $1 : \mathbf{Set}_{\text{type}} \rightarrow \mathbf{Set}_{\text{type}} \leftarrow \mathbf{Set}_{\text{type}} : 1$. More specifically, a type function morphism, from source function $x = (Y, X, x)$ to target function $x' = (Y', X', x')$, is a pair (f, g) consisting of a function $f : Y \rightarrow Y'$ between source sets and a function $g : X \rightarrow X'$ between arget sets, which satisfy the identity $f \cdot x' = x \cdot g$.

However, we do not need full generality here. Instead we restrict the definition by requiring the function g to be identity. A type function *2-cell* $\alpha : x \xrightarrow{f} x'$, with source function $Y \xrightarrow{x} X$ and target function $Y' \xrightarrow{x'} X$, has a type function $f : Y \rightarrow Y'$ that commutes with source and target functions: $f \cdot x' = x$ (Figure 4); that is, a function 2-cell is a triple $\alpha = (x, f, x')$, where (f, x') are a composable pair whose composition is $x = f \cdot x'$. Clearly, the source and target functions have a common target set.

```
(iff:set 2-cell)

(iff:function source)
(= (iff:source source) 2-cell)
(= (iff:target source) type.ftn:function)

(iff:function target)
(= (iff:source target) 2-cell)
(= (iff:target target) type.ftn:function)

(iff:function function)
(= (iff:source function) 2-cell)
(= (iff:target function) type.ftn:function)

(forall ((2-cell ?a))
  (and (type.ftn:composable-pair [(function ?a) (target ?a)])
    (= (source ?a) (type.ftn:composition [(function ?a) (target ?a)]))))
```

For any two functions (generalized elements) $x, x' \in \text{ftn}_{\text{type}}$, x *belongs to* x' , $x \sqsubseteq x'$, when there is a function 2-cell $\alpha : x \xrightarrow{R} x'$; that is, when there exists a *proof* function⁸ $p \in \text{ftn}_{\text{type}}$ such that $x = p \cdot x'$. We name the component *elements*

⁷The comma category $(1 \downarrow 1) = \mathbf{Set}_{\text{type}}^{\rightarrow}$ is called the arrow category of $\mathbf{Set}_{\text{type}}$.

⁸ p proves that x belongs to x' . In general, there may be several such proofs.

of a belonging relationship. When x' is an injection, the proof p is unique. The belongs relation is a preorder (reflexive and transitive), since 2-cells are closed under identities and composites.

```
(iff:set belonging)
(forall ((belonging ?xy)) (type.dgm.pr.mor:function-pair ?xy))
(forall ((type.ftn:function ?x) (type.ftn:function ?y))
  (<=> (belonging [?x ?y])
    (exists ((2-cell ?a)
      (and (= ?x (source ?a)) (= ?y (target ?a))))))

(iff:function element0)
(= (iff:source element0) belonging)
(= (iff:target element0) type.ftn:function)
(forall ((type.ftn:function ?x) (type.ftn:function ?y) (belonging [?x ?y]))
  (= (element0 [?x ?y]) ?x))

(iff:function element1)
(= (iff:source element1) belonging)
(= (iff:target element1) type.ftn:function)
(forall ((type.ftn:function ?x) (type.ftn:function ?y) (belonging [?x ?y]))
  (= (element1 [?x ?y]) ?y))

(forall ((type.ftn:function ?x) (type.ftn:injection ?y) (belonging [?x ?y]))
  (forall ((2-cell ?a1) (2-cell ?a2))
    (=> (and (= ?x (source ?a1)) (= ?y (target ?a1)))
      (= ?x (source ?a2)) (= ?y (target ?a2))))
  (= ?a1 ?a2))))

(forall ((type.ftn:function ?x)
  (belonging [?x ?x]))
  (forall ((type.ftn:function ?x) (type.ftn:function ?y) (type.ftn:function ?z))
    (=> (and (belonging [?x ?y]) (belonging [?y ?z]))
      (belonging [?x ?z])))

Two functions  $x, x' \in \text{ftn}_{\text{type}}$  are equivalent,  $x \equiv x'$ , when each belongs to the other,  $x \sqsubseteq x'$  and  $x' \sqsubseteq x$ . The equivalence relation is an equivalence relation (reflexive, symmetric and transitive). Two equivalent functions are isomorphic,  $x \cong x'$ , when there exists a bijection  $p \in \text{ftn}_{\text{type}}$  proving belonging. The isomorphism relation is an equivalence relation (reflexive, symmetric and transitive).
```

```
(iff:set equivalence)
(forall ((equivalence ?xy)) (type.dgm.pr.mor:function-pair ?xy))
(forall ((type.ftn:function ?x) (type.ftn:function ?y))
  (<=> (equivalence [?x ?y])
    (and (belongs [?x ?y]) (belongs [?y ?x]))))

(forall ((type.ftn:function ?x)
  (equivalence [?x ?x]))
  (forall ((type.ftn:function ?x) (type.ftn:function ?y)
    (equivalence [?x ?y]))
    (equivalence [?y ?x]))
  (forall ((type.ftn:function ?x1) (type.ftn:function ?x2) (type.ftn:function ?x3)
    (equivalence [?x1 ?x2]) (equivalence [?x2 ?x3]))
    (equivalence [?x1 ?x3]))

(iff:set isomorphism)
```

```

(forall ((isomorphism ?xy) (equivalence [?xy]))
(forall ((function ?x) (function ?y))
  (<=> (isomorphism [?x ?y])
    (exists ((2-cell ?a) (= ?x (source ?a)) (= ?y (target ?a)))
      (type.ftn:bijection (function ?a))))))

(forall ((type.ftn:function ?x)
  (isomorphism [?x ?x]))
(forall ((type.ftn:function ?x) (type.ftn:function ?y)
  (isomorphism [?x ?y]))
  (isomorphism [?y ?x]))
(forall ((type.ftn:function ?x1) (type.ftn:function ?x2) (type.ftn:function ?x3)
  (isomorphism [?x1 ?x2]) (isomorphism [?x2 ?x3]))
  (isomorphism [?x1 ?x3]))

```

Category Theory. A *pair* of type function 2-cells is *composable* when the target of the first is equal to the source of the second. We name the projection *factors* of composable pairs.

```

(iff:set composable-pair)
(forall ((composable-pair ?ab))
  (exists ((2-cell ?a) (2-cell ?b))
    (= ?ab [?a ?b])))
(forall ((2-cell ?a) (2-cell ?b))
  (<=> (composable-pair [?a ?b])
    (= (target ?a) (source ?b))))

(iff:function factor0)
(= (iff:source factor0) composable-pair)
(= (iff:target factor0) 2-cell)
(forall ((2-cell ?a) (2-cell ?b) (composable-pair [?a ?b]))
  (= (factor0 [?a ?b]) ?a))

(iff:function factor1)
(= (iff:source factor1) composable-pair)
(= (iff:target factor1) 2-cell)
(forall ((2-cell ?a) (2-cell ?b) (composable-pair [?a ?b]))
  (= (factor1 [?a ?b]) ?b))

```

The *composition* of two composable type 2-cells $\alpha : x \xRightarrow{f} y$ and $\beta : y \xRightarrow{g} z$ is a type 2-cell $\alpha \circ \beta : x \xRightarrow{f \circ g} z$. The source of the composite is the source of the first component factor, the target of the composite is the target of the second component factor, and the function of the composite is the composite of the functions of the component factors.

```

(iff:function composition)
(= (iff:source composition) composable-pair)
(= (iff:target composition) 2-cell)
(forall ((2-cell ?a) (2-cell ?b) (composable-pair [?a ?b]))
  (and (= (source (composition [?a ?b])) (source ?a))
    (= (target (composition [?a ?b])) (target ?b))
    (= (function (composition [?a ?b]))
      (type.ftn:composition [(function ?a) (function ?b)]))))

```

Composition is associative. Any three composable type 2-cells $\alpha : x \Rightarrow y$, $\beta : y \Rightarrow z$ and $\gamma : z \Rightarrow w$ satisfy the associative law $\alpha \circ (\beta \circ \gamma) = (\alpha \circ \beta) \circ \gamma$.

```
(forall ((2-cell ?a) (2-cell ?b) (2-cell ?c)
  (composable-pair [?a ?b]) (composable-pair [?b ?c]))
  (= (composition [?a (composition [?b ?c])])
    (composition [(composition [?a ?b]) ?c])))
```

The composition map $\text{mor} \times_{\text{ftn}} \text{mor} \xrightarrow{\circ} \text{mor}$ is surjective (see identity below).

```
(forall ((2-cell ?c)
  (exists ((2-cell ?a) (2-cell ?b) (composable-pair [?a ?b]))
    (= (composition [?a ?b]) ?c))))
```

For every type function $x : X \rightarrow V$, there is a unique associated *identity* type 2-cell $1_x : x \xrightarrow{1_x} x$.

```
(iff:function identity)
(= (iff:source identity) type.ftn:function)
(= (iff:target identity) 2-cell)
(forall ((type.ftn:function ?x)
  (and (= (source (identity ?x)) ?x)
    (= (target (identity ?x)) ?x)
    (= (function (identity ?x)) (type.ftn:identity (type.ftn:source ?x))))))
```

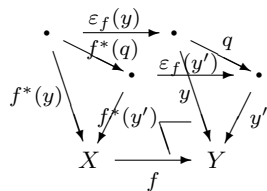
Identity satisfies two unit laws with respect to composition: composition with the identity of the source (target) of a 2-cell $\alpha : x \Rightarrow y$ returns that 2-cell: $1_x \circ \alpha = \alpha = \alpha \circ 1_y$.

```
(forall ((2-cell ?a)
  (and (= (composition [(identity (source ?a)) ?a]) ?a)
    (= ?a (composition [?a (identity (target ?a))]))))
```

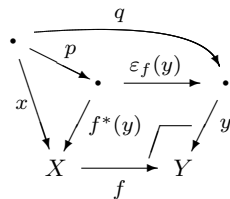
The identity map is injective $\text{ftn} \xrightarrow{1} \text{mor}$. Hence, functions can be regarded as special 2-cells that satisfy the unit laws.

```
(forall ((type.ftnt:function ?x0) (type.ftnt:function ?x1)
  (= (identity ?x0) (identity ?x1)))
  (= ?x0 ?x1))
```

Transformation. Let $f : X \rightarrow Y$ be a function. Composition with f defines an external *existential* quantification functor $\Sigma_f : \text{Set}/X \rightarrow \text{Set}/Y$ between slice categories: $x \mapsto x \cdot f$ and $(p : x \Rightarrow y) \mapsto (p : x \cdot f \Rightarrow y \cdot f)$. The existential operation is functorial: the existential quantification of the identity $1_X : X \rightarrow X$ is the identity of the existential quantification, $\Sigma_{1_X} = 1_{\text{Set}/X} : \text{Set}/X \rightarrow \text{Set}/X$; and the existential quantification of the composition $f \cdot g : X \rightarrow Z$ is the composition of the existential quantifications, $\Sigma_{f \cdot g} = \Sigma_f \cdot \Sigma_g : \text{Set}/X \rightarrow \text{Set}/Z$. Since Set_{type} has (canonical) pullbacks, the operation of pulling back along f forms an external *inverse image* functor $f^* : \text{Set}/Y \rightarrow \text{Set}/X$ between slice categories. An external universal quantification functor $\Pi_f : \text{Set}/X \rightarrow \text{Set}/Y$ can also be conceived between slice categories. The inverse (universal) image operation is also functorial. The existential functor is left adjoint to the inverse image functor and the inverse image functor is left adjoint to the universal functor $\Sigma_f \dashv (-)_f^{-1} \dashv \Pi_f$. The counit $\varepsilon_f : f^* \cdot \Sigma_f \Rightarrow 1_{\text{Set}/Y}$ for the first adjunction has the pullback projection $\varepsilon_f(y) = \pi_1 : X \times_Y \partial_0(y) \rightarrow \partial_0(y)$ as its y^{th} component. Unlike the internal transformations between subset lattices, the external transformations between complete slice categories cannot be defined at this level.



inverse image functor f^*
and counit ε_f



adjunction $\Sigma_f \dashv f^*$

$$\text{Set}/X \begin{array}{c} \xrightarrow{\Sigma_f} \\ \eta_f \dashv \varepsilon_f \\ \xleftarrow{f^*} \end{array} \text{Set}/Y$$

$$\begin{array}{ccccc} y & \xleftarrow{\varepsilon_f(y)} & \Sigma_f(f^*(y)) & & f^*(y) \\ & \searrow q & \uparrow \Sigma_f(p) = p & & \uparrow p \\ & & \Sigma_f(x) & & x \end{array}$$

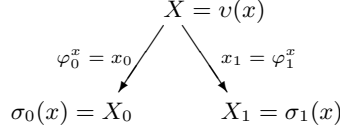


Figure 5: Span

1.4 Type Spans

Basics. A *span* is an object in the comma category

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

over the functorial ospan $\Delta : \text{Set} \rightarrow \text{Set}^2 \leftarrow \text{Set}^2 : 1$ with constant functor $\Delta : \text{Set} \rightarrow \text{Set}^2$. More specifically, a span is a triple $(X, (X_0, X_1), (x_0, x_1))$ consisting of a vertex set X , a set pair (X_0, X_1) , and a function pair $(x_0, x_1) : \Delta(X) \rightarrow (X_0, X_1)$. The first definition below expresses these observations; however, for simplicity, we use the second definition below, which is isomorphic.

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

The map $\widehat{\text{span}} \rightarrow \text{span}$ is just projection; the map $\text{span} \rightarrow \widehat{\text{span}}$ is defined by $(x_0, x_1) \mapsto (\partial_0(x_0) = \partial_0(x_1), (\partial_1(x_0), \partial_1(x_1)), (x_0, x_1))$. The spans that are axiomatized here are concrete. They can be referenced as follows.

```

(type.ftn:function ?x0)
(type.ftn:function ?x1)
(= (type.ftn:source ?x0) (type.ftn:source ?x1))
(type.ftn:function ?x)
(= ?x (type.lim.prd2.obj:pairing [?x0 ?x1]))

```

We denote a span as $x = (x_0 : X_0 \leftarrow X \rightarrow X_1 : x_1)$, picture it as in Figure 5 and reference the sets X , X_0 and X_1 as *vertex*, *domain* and *codomain*. The notion of span here is the same as the notion of span in the type limit namespace. Additional functionality is specified here.

```

(iff:set span) (= span type.lim.prd2.obj:span)
(forall ((span ?x)
  (exists ((type.ftn:function ?x0) (type.ftn:function ?x1)
    (= ?x [?x0 ?x1]))))
(forall ((type.ftn:function ?x0) (type.ftn:function ?x1)
  (<=> (span [?x0 ?x1])
    (= (type.ftn:source ?x0) (type.ftn:source ?x1))))

(iff:function function0) (= function0 type.lim.prd2.obj:function0)
(= (iff:source function0) span)
(= (iff:target function0) type.ftn:function)
(forall ((type.ftn:function ?x0) (type.ftn:function ?x1) (span [?x0 ?x1]))
  (= (function0 [?x0 ?x1]) ?x1))

(iff:function function1) (= function1 type.lim.prd2.obj:function1)

```

```

(= (iff:source function1) span)
(= (iff:target function1) type.ftn:function)
(forall ((type.ftn:function ?x0) (type.ftn:function ?x1) (span [?x0 ?x1]))
  (= (function1 [?x0 ?x1] ?x1))

(iff:function set) (iff:function vertex) (= vertex set) (= set type.lim.prd2.obj:set)
(= (iff:source set) span)
(= (iff:target set) type.set:set)
(forall ((span ?x))
  (and (= (set ?x) (type.ftn:source (function0 ?x)))
        (= (set ?x) (type.ftn:source (function1 ?x)))))

(iff:function set0) (= set0 type.lim.prd2.obj:set0)
(= (iff:source set0) span)
(= (iff:target set0) type.set:set)
(forall ((span ?x))
  (= (set0 ?x) (type.ftn:target (function0 ?x))))

(iff:function set1) (= set1 type.lim.prd2.obj:set1)
(= (iff:source set1) span)
(= (iff:target set1) type.set:set)
(forall ((span ?x))
  (= (set1 ?x) (type.ftn:target (function1 ?x))))

(iff:function set-pair) (= set-pair type.lim.prd2.obj:set-pair)
(= (iff:source set-pair) span)
(= (iff:target set-pair) type.dgm.pr.obj:set-pair)
(forall ((span ?x))
  (and (= (type.dgm.pr.obj:set0 (set-pair ?x)) (set0 ?x))
        (= (type.dgm.pr.obj:set1 (set-pair ?x)) (set1 ?x))))

(iff:function function-pair) (= function-pair type.lim.prd2.obj:function-pair)
(= (iff:source function-pair) span)
(= (iff:target function-pair) type.dgm.pr.mor:function-pair)
(forall ((span ?x))
  (and (= (type.dgm.pr.mor:source (function-pair ?x)) (type.dgm.pr.obj:constant (set ?x)))
        (= (type.dgm.pr.mor:target (function-pair ?x)) (set-pair ?x))
        (= (function-pair ?x) ?x)))

```

There is a binary *abridgment* relationship \preceq between pairs of type spans. One (smaller) type span $x = (x_0 : X_0 \leftarrow X \rightarrow X_1 : x_1)$ is the abridgment of another (larger) type span $y = (y_0 : Y_0 \leftarrow Y \rightarrow Y_1 : y_1)$, $x \preceq y$, when (1) the domain (codomain) of x is a subset of the domain (codomain) of y , $X_0 \subseteq Y_0$ ($Y_0 \subseteq Y_1$) (hence, the set-pair product of x is a subset of the set-pair product of y , $X_0 \times X_1 \subseteq Y_0 \times Y_1$), (2) the vertex of x is a subset of the vertex of y , $X \subseteq Y$, and (3) the function of x is the optimal restriction of the function of y , $\langle x \rangle \sqsubseteq \langle y \rangle$. We name the components of an abridgment relationship. The abridgment span is a partial order (reflexive, antisymmetric and transitive).

$$\begin{array}{ccc}
X & \xrightarrow{\langle x \rangle} & X_0 \times X_1 \\
\text{incl}_{X,Y} \downarrow & & \downarrow \text{incl}_{X_0 \times X_1, Y_0 \times Y_1} \\
Y & \xrightarrow{\langle y \rangle} & Y_0 \times Y_1
\end{array}$$

```
(iff:set abridgment-relation)
```

```

(forall ((abridgment-relation ?xy))
  (exists ((span ?x) (span ?y))
    (= (?xy [?x ?y])))
(forall ((span ?x) (span ?y))
  (<=> (abridgment ?x ?y)
    (and (type.set:subordinate [(set0 ?x) (set0 ?y)])
      (type.set:subordinate [(set1 ?x) (set1 ?y)])
      (type.set:subordinate [(vertex ?x) (vertex ?y)])
      (type.ftn:optimal-restriction-relation [(function ?x) (function ?y)]))))

(iff:function smaller)
(= (iff:source smaller) abridgment-relation)
(= (iff:target smaller) span)
(forall ((span ?x) (span ?y) (abridgment-relation [?x ?y]))
  (= (smaller [?x ?y]) ?x))

(iff:function larger)
(= (iff:source larger) abridgment-relation)
(= (iff:target larger) span)
(forall ((span ?x) (span ?y) (abridgment-relation [?x ?y]))
  (= (larger [?x ?y]) ?y))

(forall ((span ?x))
  (abridgment-relation [?x ?x]))
(forall ((span ?x) ?y (span ?y))
  (=> (and (abridgment-relation [?x ?y]) (abridgment-relation [?y ?x]))
    (= ?x ?y)))
(forall ((span ?x) (span ?y) (span ?z))
  (=> (and (abridgment-relation [?x ?y]) (abridgment-relation [?y ?z]))
    (abridgment-relation [?x ?z])))

```

Abridgment between two spans implies that the component functions satisfy restriction, $x_0 \sqsubseteq y_0$ and $x_1 \sqsubseteq y_1$.

```

(forall ((span ?x) (span ?y) (abridgment-relation [?x ?y]))
  (and (type.ftn:restriction [(function0 ?x) (function0 ?y)])
    (type.ftn:restriction [(function1 ?x) (function1 ?y)])))

```

Category Theory. Two type spans x and y are composable, and form a *composable pair*, when the codomain of the first is equal to the domain of the second $\sigma_1(x) = \sigma_0(y)$. We name the extent and component factors of the composable endorelation.

```

(iff:set composable-pair)
(forall ((composable-pair ?xy))
  (exists ((span ?x) (span ?y))
    (= ?xy [?x ?y])))
(forall ((span ?x) (span ?y))
  (<=> (composable-pair [?x ?y])
    (= (set1 ?x) (set0 ?y))))

(iff:function factor0)
(= (iff:source factor0) composable-pair)
(= (iff:target factor0) span)
(forall ((span ?x) (span ?y) (composable-pair [?x ?y]))
  (= (factor0 [?x ?y]) ?x))

(iff:function factor1)

```

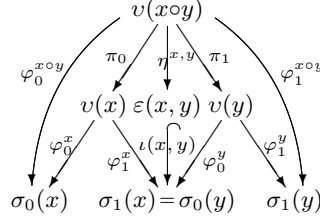


Figure 6: Composition of Spans

```
(= (iff:source factor1) composable-pair)
(= (iff:target factor1) span)
(forall ((span ?x) (span ?y) (composable-pair [?x ?y]))
  (= (factor1 [?x ?y]) ?y))
```

For any composable pair of spans (x, y) , there is an associated opspan

$$\overbrace{v(x) \xrightarrow{\varphi_1^x} \sigma_1(x) = \sigma_0(y) \xleftarrow{\varphi_0^y} v(y)}^{\Upsilon(x, y)}$$

```
(iff:function opspan)
(= (iff:source opspan) composable-pair)
(= (iff:target opspan) type.dgm.ospn.obj:opspan)
(forall ((span ?x) (span ?y) (composable-pair [?x ?y]))
  (and (= (type.dgm.ospn.obj:function0 (opspan [?x ?y])) (function1 ?x))
        (= (type.dgm.ospn.obj:function1 (opspan [?x ?y])) (function0 ?y))))
```

The *composition* of a composable pair of type spans

$$\overbrace{\sigma_0(x) \xleftarrow{\varphi_0^x} v(x) \xrightarrow{\varphi_1^x} \sigma_1(x)}^x \quad \text{and} \quad \overbrace{\sigma_0(y) \xleftarrow{\varphi_0^y} v(y) \xrightarrow{\varphi_1^y} \sigma_1(y)}^y$$

is a type span

$$\overbrace{\sigma_0(x) \xleftarrow{\varphi_0^{x \circ y}} v(x \circ y) \xrightarrow{\varphi_1^{x \circ y}} \sigma_1(y)}^{x \circ y}$$

as pictured in Figure 6. The domain of the composite is the domain of the first factor, the codomain of the composite is the codomain of the second factor, the vertex is the pullback of the associated opspan, and the component functions are composites of the pullback projections and component functions:

$$\begin{aligned} v(x \circ y) &= \lim \Upsilon(x, y) \\ \varphi_0^{x \circ y} &= \pi_0^{\Upsilon(x, y)} \cdot \varphi_0^x \\ \varphi_1^{x \circ y} &= \pi_1^{\Upsilon(x, y)} \cdot \varphi_1^y \end{aligned}$$

```
(iff:function composition)
(= (iff:source composition) composable-pair)
(= (iff:target composition) span)
(forall ((span ?x) (span ?y) (composable-pair [?x ?y]))
  (and (= (set0 (composition [?x ?y])) (set0 ?x))
        (= (set1 (composition [?x ?y])) (set1 ?y))))
```

```

(= (vertex (composition [?x ?y])) (type.lim.pbk.obj:pullback (opspan [?x ?y])))
(= (function0 (composition [?x ?y]))
  (type.ftn:composition
   [(type.lim.pbk.obj:projection0 (opspan [?x ?y])) (function0 ?x)]))
(= (function1 (composition [?x ?y]))
  (type.ftn:composition
   [(type.lim.pbk.obj:projection1 (opspan [?x ?y])) (function1 ?y)])))

```

Currently, pullbacks in the type namespace are concrete. Hence, the extent of the composition is $v(x \circ y) = \{(a, b) \mid a \in v(x), b \in v(y), \varphi_1^x(a) = \varphi_0^y(b)\}$, and the function components are $\varphi_0^{x \circ y}(a, b) = \varphi_0^x(a)$ and $\varphi_1^{x \circ y}(a, b) = \varphi_1^y(b)$.

```

(forall ((span ?x) (span ?y) (composable-pair [?x ?y]))
  (and (forall ((vertex (composition [?x ?y])) ?ab)
    (exists ((vertex ?x) ?a) ((vertex ?y) ?b)
      (= ?ab [?a ?b])))
    (forall ((vertex ?x) ?a) ((vertex ?y) ?b)
      (<=> ((vertex (composition [?x ?y])) [?a ?b])
        (= ((function1 ?x) ?a) ((function0 ?y) ?b)))))

(forall ((span ?x) (span ?y) (composable-pair [?x ?y])
  ((vertex ?x) ?a) ((vertex ?y) ?b) ((vertex (composition [?x ?y])) [?a ?b]))
  (and (= ((function0 (composition [?x ?y])) [?a ?b]) ((function0 ?x) ?a))
    (= ((function1 (composition [?x ?y])) [?a ?b]) ((function1 ?y) ?b))))

```

Composition is associative up to isomorphism.

```

(forall ((span ?x) (span ?y) (span ?z)
  (composable-pair [?x ?y]) (composable-pair ?y ?z))
  (isomorphism [(composition [?x (composition [?y ?z])]
    (composition [(composition [?x ?y]) ?z]))])

```

Composition is surjective (see identity properties below).

```

(forall ((span ?z)
  (exists ((span ?x) (span ?y) (composable-pair [?x ?y]))
    (= (composition [?x ?y]) ?z)))

```

For every type set X , there is an associated *identity* type span

$$\overbrace{X \xleftarrow{1_X} X \xrightarrow{1_X} X}^{1_X}$$

with X as domain and codomain. Its vertex is X and its component functions are $1_X : X \rightarrow X$.

```

(iff:function identity)
(= (iff:source identity) type.set:set)
(= (iff:target identity) span)
(forall ((type.set:set ?X)
  (and (= (set0 (identity ?X)) ?X)
    (= (set1 (identity ?X)) ?X)
    (= (vertex (identity ?X)) ?X)
    (= (function0 (identity ?X)) (type.ftn:identity ?X))
    (= (function1 (identity ?X)) (type.ftn:identity ?X))))

```

The identity satisfies up to isomorphism two unit laws with respect to composition.

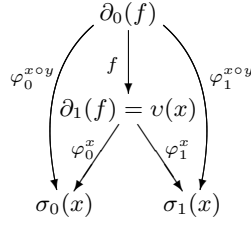


Figure 7: Combination of Function and Span

```
(forall ((span ?x))
  (and (isomorphism [(composition [(identity (set0 ?x)) ?x]) ?x])
    (isomorphism [?x (composition [?x (identity (set1 ?x))])]))))
```

Identity is injective; hence, sets can be regarded as special spans that satisfy the unit laws.

```
(forall ((type.set:set ?X0) (type.set:set ?X1)
  (= (identity ?X0) (identity ?X1)))
  (= ?X0 ?X1))
```

A type function f and a type span x are combinable, and form a *combinable pair*, when the functions $(f, \langle x \rangle)$ are composable, and form a composable pair of functions; that is, when the target of the function f is equal to the vertex of the span x , $\partial_1(f) = v(x)$. We name the extent and component factors of the combinable endorelation.

```
(iff:set combinable-pair)
(forall ((combinable-pair ?fx))
  (exists ((type.ftn:function ?f) (span ?x))
    (= ?fx [?f ?x])))
(forall ((type.ftn:function ?f) (span ?x))
  (<=> (combinable-pair [?f ?x])
    (type.ftn:composable-pair [?f (function ?x)])))
(forall ((type.ftn:function ?f) (span ?x))
  (<=> (combinable-pair [?f ?x])
    (= (type.ftn:target ?f) (vertex ?x))))

(iff:function component0)
(= (iff:source component0) combinable-pair)
(= (iff:target component0) type.ftn:function)
(forall ((type.ftn:function ?f) (span ?x) (combinable-pair [?f ?x]))
  (= (component0 [?f ?x]) ?f))

(iff:function component1)
(= (iff:source component1) combinable-pair)
(= (iff:target component1) span)
(forall ((type.ftn:function ?f) (span ?x) (combinable-pair [?f ?x]))
  (= (component1 [?f ?x]) ?x))
```

The *combination* of a combinable pair

$$\overbrace{\partial_0(f) \xrightarrow{f} \partial_1(f)}^f \quad \text{and} \quad \overbrace{\sigma_0(x) \xleftarrow{\varphi_0^x} v(x) \xrightarrow{\varphi_1^x} \sigma_1(x)}^x$$

is a type span

$$\overbrace{\sigma_0(x) \xleftarrow{\varphi_0^{f \circ x}} v(f \circ x) \xrightarrow{\varphi_1^{f \circ x}} \sigma_1(y)}^{f \circ x}$$

as pictured in Figure 7. The function of the combination is the function composition $\langle f \diamond x \rangle = f \cdot \langle x \rangle$. The domain (codomain) of the combination is the domain (codomain) of the span, the vertex of the combination is the source of the function, and the component functions of the combination are the composites of the function and component functions of the span:

$$\begin{aligned}\sigma_0(f \diamond x) &= \sigma_0(x) \\ \sigma_1(f \diamond x) &= \sigma_1(x) \\ v(f \diamond x) &= \partial_0(f) \\ \varphi_0^{f \diamond x} &= f \cdot \varphi_0^x \\ \varphi_1^{f \diamond x} &= f \cdot \varphi_1^x.\end{aligned}$$

```
(iff:function combination)
(= (iff:source combination) combinable-pair)
(= (iff:target combination) span)
(forall ((type.ftn:function ?f) (span ?x) (combinable-pair [?f ?x]))
  (and (= (function (combination [?f ?x])) (type.ftn:composition [?f (function ?x)]))
    (= (set0 (combination [?f ?x])) (set0 ?x))
    (= (set1 (combination [?f ?x])) (set1 ?x))
    (= (vertex (combination [?f ?x])) (type.ftn:source ?f))
    (= (function0 (combination [?f ?x])) (type.ftn:composition [?f (function0 ?x)]))
    (= (function1 (combination [?f ?x])) (type.ftn:composition [?f (function1 ?x)]))))
```

Combination satisfies a mixed associative law: $f_2 \diamond (f_1 \diamond x) = (f_2 \cdot f_1) \diamond x$ for composable pairs (f_2, f_1) and combinable pairs (f_1, x) , since $\langle f_2 \diamond (f_1 \diamond x) \rangle = f_2 \cdot \langle f_1 \diamond x \rangle = f_2 \cdot (f_1 \cdot \langle x \rangle) = (f_2 \cdot f_1) \cdot \langle x \rangle = \langle (f_2 \cdot f_1) \diamond x \rangle$.

```
(forall ((type.ftn:function ?f2) (type.ftn:function ?f1) (span ?x)
  (composable-pair [?f2 ?f1]) (combinable-pair [?f1 ?x]))
  (= (combination [?f2 (combination [?f1 ?x])])
    (combination [(type.ftn:composition [?f2 ?f1]) ?x])))
```

Factorization. There are special kinds of type spans. A type span $x = (x_0 : X_0 \leftarrow X \rightarrow X_1 : x_1)$ is an *injection* (*surjection*, *bijection*) when its function $\langle x \rangle : X \rightarrow X_0 \times X_1$ is an injection (surjection, bijection). Injections are essentially relational spans, and bijections are essentially binary product spans of set pairs.

```
(iff:set injection)
(forall ((injection ?x) (span ?x))
  (forall ((span ?x)
    (<=> (injection ?x)
      (type.ftn:injection (function ?x))))))
(forall ((span ?x)
  (<=> (injection ?x)
    (exists (?r (type.rel:relation ?r))
      (= ?x (type.rel:span ?r))))))

(iff:set surjection)
(forall ((surjection ?x) (span ?x))
  (forall ((span ?x)
    (<=> (surjection ?x)
      (type.ftn:surjection (function ?x))))))

(iff:set bijection)
(forall ((bijection ?x) (span ?x))
  (forall ((span ?x)
```

```

(<=> (bijection ?x)
      (type.ftn:bijection (function ?x)))
(forall ((span ?x))
  (<=> (bijection ?x)
        (exists (?X01 (type.dgm.pr.obj:set-pair ?X01))
          (isomorphism [?x (type.lim.prd2.obj:span ?X01)]))))

```

A span is a bijection when it is both an injection and a surjection.

```

(forall ((span ?x))
  (<=> (bijection ?x)
        (and (injection ?x) (surjection ?x))))

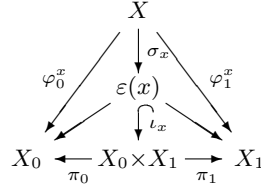
```

Injections, surjections and bijections are closed under combination with function injections, function surjections and function bijections.

```

(forall ((type.ftn:injection ?f) (injection ?x) (combinable-pair [?f ?x]))
  (injection (combination [?f ?x])))
(forall ((type.ftn:surjection ?f) (surjection ?x) (combinable-pair [?f ?x]))
  (surjection (combination [?f ?x])))
(forall ((type.ftn:bijection ?f) (bijection ?x) (combinable-pair [?f ?x]))
  (bijection (combination [?f ?x])))

```



For any type span $x = (x_0 : X_0 \leftarrow X \rightarrow X_1 : x_1)$, there is a *extent* (or *range*) type set $\varepsilon(x) = \text{ext}(x) \subseteq X_0 \times X_1$, defined as the range of the function $\langle x \rangle : X \rightarrow X_0 \times X_1$.

```

(iff:function extent) (iff:function range) (= range extent)
(= (iff:source range) span)
(= (iff:target range) type.set:set)
(= (range ?x) (type.ftn:range (function ?x)))

```

Pointwise this is $\varepsilon(x) = \{(a_0, a_1) \in X_0 \times X_1 \mid \exists a \in X \langle x \rangle(a) = (a_0, a_1)\}$.

```

(forall ((span ?x))
  (subset-relation [(range ?x) (type.lim.prd2.obj:product (set-pair ?x)])))
(forall ((span ?x) ((set0 ?x) ?a0) ((set1 ?x) ?a1))
  (<=> ((range ?x) [?a0 ?a1])
        (exists ((vertex ?x) ?a)
          (= [?a0 ?a1] ((function ?x) ?a))))

```

For any type span $x = (x_0 : X_0 \leftarrow X \rightarrow X_1 : x_1)$, the *injective factor* $\iota_x : \varepsilon(x) \rightarrow X_0 \times X_1$ (*surjective factor* $\sigma_x : X \rightarrow \varepsilon(x)$) is the injective factor (surjective factor) of the function $\langle x \rangle : X \rightarrow X_0 \times X_1$.

```

(iff:function injective-factor)
(= (iff:source injective-factor) span)
(= (iff:target injective-factor) type.ftn:function)
(forall ((span ?x))
  (and (= (type.ftn:source (injective-factor ?x)) (range ?x))
        (= (type.ftn:target (injective-factor ?x))
           (type.lim.prd2.obj:product (set-pair ?x)))))

```

```

      (type.lim.prd2.obj:product (set-pair ?x))
      (type.ftn:injection (injective-factor ?x))
      (= (injective-factor ?x) (type.ftn:injective-factor (function ?x))))

(iff:function surjective-factor)
(= (iff:source surjective-factor) span)
(= (iff:target surjective-factor) type.ftn:function)
(forall ((span ?x)
  (and (= (type.ftn:source (surjective-factor ?x)) (source ?x))
        (= (type.ftn:target (surjective-factor ?x)) (range ?x))
        (type.ftn:surjection (surjective-factor ?x))
        (= (surjective-factor ?x) (type.ftn:surjective-factor (function ?x)))))

```

Pointwise, the injective factor is the inclusion of the range into the product of the set pair of x , and the surjective factor is the restriction of the function $\langle x \rangle$ to the range.

```

(forall ((span ?x) ((set0 ?x) ?a0) ((set1 ?x) ?a1) ((range ?x) [?a0 ?a1]))
  (= ((injective-factor ?x) [?a0 ?a1]) [?a0 ?a1]))

(forall ((span ?x) ((vertex ?x) ?a))
  (= ((surjective-factor ?x) ?a) ((function ?x) ?a)))

```

The function $\langle x \rangle : X \rightarrow X_0 \times X_1$ is the composition of its surjective factor and injective factor: $\langle x \rangle = \sigma_x \cdot \iota_x : X \rightarrow \varepsilon(x) \rightarrow X_0 \times X_1$.

```

(forall ((span ?x)
  (= (function ?x)
    (type.ftn:composition [(surjective-factor ?x) (injective-factor ?x)])))

```

The (surjective-factor, injective-factor) pair forms a surjection-injection “left factorization system” for type sets, functions and type spans; that is, if the function $\langle x \rangle : X \rightarrow X_0 \times X_1$ of a type span $x = (x_0 : X_0 \leftarrow X \rightarrow X_1 : x_1)$ is the composition of a surjection with an injection, $\langle x \rangle = e \cdot m : X \rightarrow Z \rightarrow X_0 \times X_1$, then there is a (unique) “diagonal” bijection $d : \varepsilon(x) \rightarrow Z$, such that $\sigma_x \cdot d = e$ and $d \cdot m = \iota_x$.

$$\begin{array}{ccc}
 X & \xrightarrow{\sigma_x} & \varepsilon(x) \\
 e \downarrow & \swarrow d & \downarrow \iota_x \\
 Z & \xrightarrow{m} & X_0 \times X_1
 \end{array}$$

```

(forall ((span ?x) (type.ftn:surjection ?e) (type.ftn:injection ?m)
  (= (function ?x) (type.ftn:composition [?e ?m])))
  (and (exists ((type.ftn:bijection ?d)
    (= (type.ftn:source ?d) (range ?x))
    (= (type.ftn:target ?d) (type.ftn:target ?e)))
    (and (= (type.ftn:composition [(surjective-factor ?x) ?d]) ?e)
          (= (type.ftn:composition [?d ?m]) (injective-factor ?x))))
    (forall ((bijection ?d1) (bijection ?d2)
      (= (type.ftn:source ?d1) (range ?x))
      (= (type.ftn:source ?d2) (range ?x))
      (= (type.ftn:target ?d1) (type.ftn:target ?e))
      (= (type.ftn:target ?d2) (type.ftn:target ?e))
      (= (type.ftn:composition [(surjective-factor ?x) ?d1]) ?e)

```

```

      (= (type.ftn:composition [(surjective-factor ?x) ?d2]) ?e)
      (= (type.ftn:composition [?d1 ?m]) (injective-factor ?x))
      (= (type.ftn:composition [?d2 ?m]) (injective-factor ?x)))
    (= ?d1 ?d2)))

```

A surjection is a span whose injective factor is a bijection. An injection is a span whose surjective factor is a bijection. An injection is a span that is the embedding of some relation.

```

(forall ((span ?x))
  (<=> (surjection ?x)
    (type.ftn:bijection (injective-factor ?x))))

(forall ((span ?x))
  (<=> (injection ?x)
    (type.ftn:bijection (surjective-factor ?x))))

(forall ((span ?x))
  (<=> (injection ?x)
    (type.spn.mor:isomorphism [?x (type.rel:span (relation ?x))])))

(forall ((span ?x))
  (<=> (injection ?x)
    (exists (?r (type.rel:relation ?r))
      (type.spn.mor:isomorphism [?x (type.rel:span ?r)]))))

```

Let $x = (x_0 : X_0 \leftarrow X \rightarrow X_1 : x_1)$ be a type span. The *kernel* (*coimage*, *coequalizer*) of x is the kernel (coimage, coequalizer) of the function $\langle x \rangle : X \rightarrow X_0 \times X_1$ ⁹.

```

(iff:function kernel)
(= (iff:source kernel) span)
(= (iff:target kernel) type.rel:equivalence-relation)
(forall ((span ?x))
  (and (= (type.rel:set (kernel ?x)) (vertex ?x))
    (= (kernel ?x) (type.ftn:kernel (function ?x)))))

(iff:function coimage)
(= (iff:source coimage) span)
(= (iff:target coimage) type.set:set)
(forall ((span ?x))
  (= (coimage ?x) (type.ftn:coimage (function ?x))))

(iff:function coequalizer)
(= (iff:source coequalizer) span)
(= (iff:target coequalizer) type.ftn:function)
(forall ((span ?f))
  (and (= (type.ftn:source (coequalizer ?x)) (vertex ?x))
    (= (type.ftn:target (coequalizer ?x)) (coimage ?x))
    (type.ftn:surjection (coequalizer ?x))
    (= (coequalizer ?x) (type.ftn:coequalizer (function ?x)))))

```

⁹Pointwise, (1) the kernel of x is the equivalence relation $\ker_x = \equiv_x$ on the vertex set X defined by $a \equiv_x b$ iff $\langle x \rangle(a) = (x_0(a), x_1(a)) = (x_0(b), x_1(b)) = \langle x \rangle(b)$ for all vertex pairs $a, b \in X$ (hence, the kernel of the span x is the intersection $\equiv_x = \equiv_{x_0} \cap \equiv_{x_1}$ of the kernels of the component functions x_0 and x_1); (2) the coimage of x is the quotient of the kernel $\text{coim}_x = X/\equiv_x = \{ [a]_{\equiv_x} \mid a \in X \}$, where $[a]_x = \{ b \in X \mid \langle x \rangle(b) = \langle x \rangle(a) \}$ is the equivalence class of a with respect to the kernel of x ; and (3) the coequalizer of x is the canon of the kernel $\text{coeq}_x = [-]_x = [-]_{\equiv_x} : X \rightarrow X/\equiv_x$.

The function $\langle x \rangle : X \rightarrow X_0 \times X_1$ of any type span $x = (x_0 : X_0 \leftarrow X \rightarrow X_1 : x_1)$ respects its kernel $\pi_0^{\equiv_x} \cdot \langle x \rangle = \pi_1^{\equiv_x} \cdot \langle x \rangle$, and hence factors through its coimage $\langle x \rangle = [-]_x \cdot \wr_x$ for some (injective) span $\text{apply}_x = \wr_x : \text{coim}_x \rightarrow X_0 \times X_1$. This factor, called the *coapplication* of x , is defined by $\wr_f([a]_{\equiv_x}) = \langle x \rangle(a) = (x_0(a), x_1(a))$ for all $a \in X$.

$$\begin{array}{ccc} \text{ext}(\equiv_x) & \begin{array}{c} \xrightarrow{\pi_0} \\ \xrightarrow{\pi_1} \end{array} & X & \xrightarrow{\langle x \rangle} & X_0 \times X_1 \\ & & \searrow [-]_x & & \nearrow \wr_x \\ & & & & X/\equiv_x \end{array}$$

```
(iff:function coapplication)
(= (iff:source coapplication) span)
(= (iff:target coapplication) type.ftn:function)
(forall ((span ?x))
  (and (= (type.ftn:source (coapplication ?x)) (coimage ?x))
        (= (type.ftn:target (coapplication ?x)) (type.lim.prd2.obj:product (set-pair ?x)))
        (type.ftn:injection (coapplication ?x))
        (= (function ?x) (type.ftn:composition [(coequalizer ?x) (coapplication ?x)]))))
```

The (coequalizer, coapplication) pair forms a surjection-injection “left factorization system” for type sets, type functions and type spans; that is, if the function $\langle x \rangle : X \rightarrow X_0 \times X_1$ of a type span $x = (x_0 : X_0 \leftarrow X \rightarrow X_1 : x_1)$ is the composition of a surjection with an injection, $\langle x \rangle = e \cdot m : X \rightarrow Z \rightarrow X_0 \times X_1$, then there is a (unique) “diagonal” bijection $d : X/\equiv_f \rightarrow Z$ such that $[-]_x \cdot d = e$ and $d \cdot m = \wr_x$.

$$\begin{array}{ccc} X & \xrightarrow{[-]_f} & X/\equiv_f \\ e \downarrow & \swarrow d & \downarrow \wr_f \\ Z & \xrightarrow{m} & X_0 \times X_1 \end{array}$$

This implies that the coimage is naturally isomorphic to the range (image), $\text{coim}_x \cong \text{rng}_x$; specifically, for any source element of $a \in X$, the equivalence class $[a]_{\equiv_x} \in \text{coim}_x$ corresponds to the image element $\langle x \rangle(a) = (x_0(a), x_1(a)) \in \text{rng}_x$.

Conversion. Any type span

$$\overbrace{\sigma_0(x) \xleftarrow{\varphi_0^x} v(x) \xrightarrow{\varphi_1^x} \sigma_1(x)}^x$$

has an *opposite* type span

$$\overbrace{\sigma_1(x) \xleftarrow{\varphi_1^x} v(x) \xrightarrow{\varphi_0^x} \sigma_0(x)}^{x^\infty}$$

```
(iff:function opposite)
(= (iff:source opposite) span)
(= (iff:target opposite) span)
```

```

(forall ((span ?x))
  (and (= (set0 (opposite ?x)) (set1 ?x))
        (= (set1 (opposite ?x)) (set0 ?x))
        (= (vertex (opposite ?x)) (vertex ?x))
        (= (function0 (opposite ?x)) (function1 ?x))
        (= (function1 (opposite ?x)) (function0 ?x))))

```

The opposite is an pseudo-involution: $r^{\alpha\alpha} = r$, $(r \circ s)^{\alpha} \cong s^{\alpha} \circ r^{\alpha}$, and $1_X^{\alpha} = 1_X$.

```

(forall ((span ?x))
  (= (opposite (opposite ?x)) ?x))

(forall ((span ?x) (span ?y) (composable-pair [?x ?y]))
  (isomorphism [(opposite (composition [?x ?y]))
                (composition [(opposite ?y) (opposite ?x)])]))

(forall ((type.set:set ?X))
  (= (opposite (identity ?X)) (identity ?X)))

```

Any type span $x = (X_0 \xleftarrow{x_0} X \xrightarrow{x_1} X_1)$ has an associated *function* $\langle x \rangle = \langle x_0, x_1 \rangle : X \rightarrow X_0 \times X_1$, which is the product pairing of the function pair.

```

(iff:function function)
(= (iff:source function) span)
(= (iff:target function) type.ftn:function)
(forall ((span ?x))
  (and (= (type.ftn:source (function ?x)) (vertex ?x))
        (= (type.ftn:target (function ?x)) (type.lim.prd2.obj:product (set-pair ?x)))
        (= (function ?x) (type.lim.prd2.obj:pairing ?x))))

```

Any type span $x = (X_0 \xleftarrow{x_0} X \xrightarrow{x_1} X_1)$ has an associated *relation* $\text{rel}(x) = \tilde{x} : X_0 \rightarrow X_0$, whose set pair is the set pair of the span, whose extent is the extent of the span, and whose injection is the injective factor of the span.

```

(iff:function relation)
(= (iff:source relation) span)
(= (iff:target relation) type.rel:relation)
(forall ((span ?x))
  (and (= (type.rel:set0 (relation ?x)) (set0 ?x))
        (= (type.rel:set1 (relation ?x)) (set1 ?x))
        (= (type.rel:extent (relation ?x)) (extent ?x))
        (= (type.rel:function (relation ?x)) (injective-factor ?x))))

```

The relation of a span composition is the relation composition of the spans $\text{rel}(x \circ y) = \text{rel}(x) \circ \text{rel}(y)$. The relation of the span identity is the relation identity $\text{rel}(1_X) = 1_X$. The relation of the opposite of a span is the opposite of the relation of the span $\text{rel}(x^{\alpha}) = \text{rel}(x)^{\alpha}$.

```

(forall ((span ?x) (span ?y) (composable-pair [?x ?y]))
  (= (relation (composition [?x ?y]))
     (type.rel:composition [(relation ?x) (relation ?y)])))

(forall ((type.set:set ?X))
  (= (relation (identity ?X)) (type.rel:identity ?X)))

(forall ((span ?x))
  (= (relation (opposite ?x)) (type.rel:opposite (relation ?x))))

```

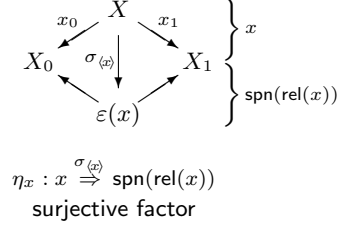


Figure 8: Unit

Any type span $x = (X_0 \xleftarrow{x_0} X \xrightarrow{x_1} X_1)$ is linked to the span of its relation by the *unit* span 2-cell $\eta_x : x \xrightarrow{\sigma(x)} \text{spn}(\text{rel}(x))$, where the function $\sigma(x) : X \rightarrow \varepsilon(x)$ is the surjective factor of x (Figure 8).

```

(iff:function unit)
(= (iff:source unit) span)
(= (iff:target unit) type.spn.mor:2-cell)
(forall ((span ?x))
  (and (= (type.spn.mor:source (unit ?r)) ?x)
        (= (type.spn.mor:target (unit ?r)) (type.rel:span (relation ?x)))
        (= (type.spn.mor:function (unit ?r)) (surjective-factor ?x))))

```

For any two spans $y = (X_0 \xleftarrow{y_0} Y \xrightarrow{y_1} X_1)$ and $z = (X_0 \xleftarrow{z_0} Z \xrightarrow{z_1} X_1)$ that share common domain and codomain sets, there is a fiber product span $y \times z = (X_0 \leftarrow Y \times_{X_0 \times X_1} Z \rightarrow X_1)$ whose vertex and projection functions are defined in terms of the pullback of the pairing functions. The pullback projections are the functions of 2-cells $\pi_0 : y \times z \xrightarrow{\pi_0} y$ and $\pi_1 : y \times z \xrightarrow{\pi_1} z$, showing that the fiber product belongs to each component: $y \times z \in y$ and $y \times z \in z$. Any other span that belongs to y and z also belongs to the fiber product $y \times z$.

```

(forall ((span ?y) (span ?z) (= (set-pair ?y) (set-pair ?z)))
  (and (= (set (product [?y ?z])) (type.lim.pbk.obj:pullback [(function ?y) (function ?z)]))
        (= (function0 (product [?y ?z]))
            (type.ftn:composition
              [(type.lim.pbk.obj:projection0 [(function ?y) (function ?z)] (function0 ?y)]))
          (= (function1 (product [?y ?z]))
              (type.ftn:composition
                [(type.lim.pbk.obj:projection1 [(function ?y) (function ?z)] (function1 ?z)]))))))
  (forall ((span ?w) (= (set-pair ?w) (set-pair ?y)))
    (=> (and (type.spn.mor:belonging [?w ?y]) (type.spn.mor:belonging [?w ?z]))
         (type.spn.mor:belonging [?w (product [?y ?z])]))))

```

Instances. For any set pair (X_0, X_1) , the empty set and counique functions form an *initial* (relational) span $0_{X_0, X_1} = (X_0 \xleftarrow{\hat{i}_{X_0}} \emptyset \xrightarrow{\hat{i}_{X_1}} X_1)$ with $\langle 0_{X_0, X_1} \rangle =$

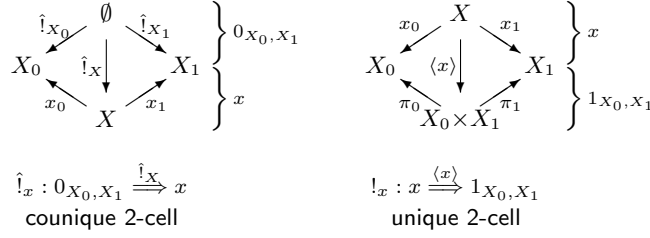


Figure 9: Counique and Unique Span 2-Cells

$0_{X_0 \times X_1}$, and the binary product and projections form a *terminal* (relational) span $1_{X_0, X_1} = (X_0 \xleftarrow{\pi_0} X_0 \times X_1 \xrightarrow{\pi_1} X_1)$ with $\langle 1_{X_0, X_1} \rangle = 1_{X_0 \times X_1}$.

```
(iff:function initial)
(= (iff:source initial) type.dgm.pr.obj:set-pair)
(= (iff:target initial) span)
(forall ((type.dgm.pr.obj:set-pair ?X01))
  (and (= (function0 (initial ?X01)) (type.set:counique (type.dgm.pr.obj:set0 ?X01)))
        (= (function1 (initial ?X01)) (type.set:counique (type.dgm.pr.obj:set1 ?X01)))
        (= (vertex (initial ?X01)) type.set:zero)
        (= (set0 (initial ?X01)) (type.dgm.pr.obj:set0 ?X01))
        (= (set1 (initial ?X01)) (type.dgm.pr.obj:set1 ?X01))
        (= (set-pair (initial ?X01)) ?X01)))

(iff:function terminal)
(= (iff:source terminal) type.dgm.pr.obj:set-pair)
(= (iff:target terminal) span)
(forall ((type.dgm.pr.obj:set-pair ?X01))
  (and (= (function0 (terminal ?X01)) (type.lim.prd2.obj:projection0 ?X01))
        (= (function1 (terminal ?X01)) (type.lim.prd2.obj:projection1 ?X01))
        (= (vertex (terminal ?X01)) (type.lim.prd2.obj:product ?X01))
        (= (set0 (terminal ?X01)) (type.dgm.pr.obj:set0 ?X01))
        (= (set1 (terminal ?X01)) (type.dgm.pr.obj:set1 ?X01))
        (= (set-pair (terminal ?X01)) ?X01)))
```

Let $x = (X_0 \xleftarrow{x_0} X \xrightarrow{x_1} X_1)$ be any type span.

- There is a *counique* type span 2-cell $\hat{!}_x : 0_{X_0, X_1} \rightrightarrows x$ (left side of Figure 9), whose source is the initial span $0_{X_0, X_1}$, whose target is x and whose function is $\hat{!}_X : \emptyset \rightarrow X$. This is the unique type span 2-cell from $0_{X_0, X_1}$ to x . Hence, $0_{X_0, X_1}$ is the initial object in the slice category $\text{Set}/(X_0, X_1)$. Here, $\langle \hat{!}_x \rangle = \hat{!}_{\langle x \rangle}$.
- There is a *unique* type span 2-cell $!_x : x \rightrightarrows 1_{X_0, X_1}$ (right side of Figure 9), whose source is x , whose target is the terminal span $1_{X_0, X_1}$ and whose function is $\langle x \rangle$. This is the unique type span 2-cell from x to $1_{X_0, X_1}$. Hence, $1_{X_0, X_1}$ is the terminal object in the slice category $\text{Set}/(X_0, X_1)$. Here, $\langle !_x \rangle = !_{\langle x \rangle}$. Note: $0_{X_0, X_1}$ is the (span of the) bottom relation in the fiber over (X_0, X_1) , and $1_{X_0, X_1}$ is the (span of the) top relation in the fiber over (X_0, X_1) .

```

(iff:function counique)
(= (iff:source counique) type.spn:span)
(= (iff:target counique) type.spn.mor:2-cell)
(forall ((span ?x))
  (and (= (type.spn.mor:source (counique ?x)) (initial (set-pair ?x)))
        (= (type.spn.mor:target (counique ?x)) ?x)
        (= (type.spn.mor:function (counique ?x)) (type.set:counique (vertex ?x))))
    (forall ((type.spn.mor:2-cell ?a)
              (= (type.spn.mor:source ?a) (initial (set-pair ?x)))
              (= (type.spn.mor:target ?a) ?x))
      (= ?a (counique ?x))))))

(iff:function unique)
(= (iff:source unique) span)
(= (iff:target unique) type.spn.mor:2-cell)
(forall ((span ?x))
  (and (= (type.spn.mor:source (unique ?x)) ?x)
        (= (type.spn.mor:target (unique ?x)) (terminal (set-pair ?x)))
        (= (type.spn.mor:function (unique ?x)) (function ?x)))
    (forall ((type.spn.mor:2-cell ?a)
              (= (type.spn.mor:source ?a) ?x)
              (= (type.spn.mor:target ?a) (terminal (set-pair ?x))))
      (= ?a (unique ?x))))))

```

1.4.1 Type Endospans

Basics. The set of all type *endospans* is an IFF set. Type endospans are special type spans, whose domain and codomain sets are identical.

```
(iff:set endospan)
(forall ((endospan ?x)) (span ?x))
(forall ((span ?x))
  (<=> (endospan ?x)
    (= (set0 ?x) (set1 ?x))))
```

There is a common component type *set*.

```
(iff:function set)
(= (iff:source set) endospan)
(= (iff:target set) type.set:set)
(forall (?x (endospan ?x))
  (and (= (set ?x) (set0 ?x))
    (= (set ?x) (set1 ?x))))
```

Order Theory. We have predicates to express the fact that a type endospan is *reflexive*, *transitive* or *symmetric*. These are predicates (adjectives) that refer to (modify) endospans. Let $x = (X \xleftarrow{\partial_0} Y \xrightarrow{\partial_1} X)$ be a type endospan.

- x is *reflexive* when $1_X \in x$; that is, when the identity endospan belongs to x ; that is, when there is a span 2-cell $1_X \xrightarrow{\iota} x$ from 1_X to x . that is, when there exists a proof function $\iota : X \rightarrow Y$ such that $\iota \cdot \partial_0 = 1_X$ and $\iota \cdot \partial_1 = 1_X$.
- x is *transitive* when $x \circ x \in x$; that is, when the composition belongs to x ; that is, when there is a span 2-cell $x \circ x \xrightarrow{\mu} x$ from $x \circ x$ to x . that is, when there exists a proof function $\mu : Y \times_X Y \rightarrow Y$ such that $\mu \cdot \partial_0 = \pi_0 \cdot \partial_0$ and $\mu \cdot \partial_1 = \pi_1 \cdot \partial_1$.
- x is *symmetric* when $x^\circ \in x$; that is, the transpose belongs to x . Since the relation conversion preserves transpose, $\text{rel}(x)$ is symmetric; that is, the relation of a symmetric span is a symmetric relation $\text{rel}(x)^\circ = \text{rel}(x^\circ) \subseteq \text{rel}(x)$.

```
(iff:set reflexive-span)
(forall ((reflexive-span ?x)) (endospan ?x))
(forall ((endospan ?x))
  (<=> (reflexive-span ?x)
    (type.spn.mor:belonging [(identity (set ?x)) ?x])))
(forall ((endospan ?x))
  (=> (reflexive-span ?x) (type.rel:reflexive-relation (relation ?x))))
```

```
(iff:set transitive-span)
(forall ((transitive-span ?x)) (endospan ?x))
(forall ((endospan ?x))
  (<=> (transitive-span ?x)
    (type.spn.mor:belonging [(composition [?x ?x]) ?x])))
(forall ((endospan ?x))
  (=> (transitive-span ?x) (type.rel:transitive-relation (relation ?x))))
```

```

(iff:set symmetric-span)
(forall ((symmetric-span ?x)) (endospan ?x))
(forall ((endospan ?x))
  (<=> (symmetric-span ?x)
    (type.spn.mor:belonging [(opposite ?x) ?x])))
(forall ((endospan ?x))
  (=> (symmetric-span ?x) (type.rel:symmetric-relation (relation ?x))))

```

We can declare special type endospans called proto-categories and equivalence spans. A *proto-category* $x = (X \xleftarrow{\partial_0} Y \xrightarrow{\partial_1} X)$ with object set X and morphism set Y , is a reflexive and transitive endospan. The proof function $\iota : X \rightarrow Y$ is a proto-identity function and the proof function $\mu : Y \times_X Y \rightarrow Y$ is a proto-composition function. Since belonging implies inclusion and the relation conversion preserves identity and composition, $\text{rel}(x)$ is reflexive and transitive; that is, the relation of a proto-category (preorder span) is a preorder relation. Any category is a proto-category that satisfies associative and unit laws. An *equivalence span* is a reflexive, symmetric and transitive span.

```

(iff:set proto-category)
(forall ((proto-category ?x)) (endospan ?x))
(forall ((endospan ?x))
  (<=> (proto-category ?x)
    (and (reflexive-span ?x) (transitive-span ?x))))
(forall ((endospan ?x))
  (=> (proto-category ?x) (type.rel:preorder (relation ?x))))

(iff:set equivalence-span)
(forall ((equivalence-span ?x)) (preorder ?x))
(forall ((preorder ?x))
  (<=> (equivalence-span ?x)
    (symmetric-span ?x)))
(forall ((endospan ?x))
  (=> (equivalence-span ?x) (type.rel:equivalence-relation (relation ?x))))

```

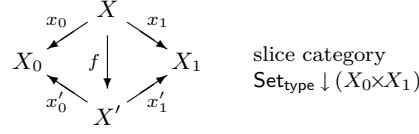


Figure 10: Span 2-Cell

1.4.2 Span Morphisms

Basics. A type span *morphism* is a morphism in the comma category

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

over the functorial ospan $\Delta : \text{Set}_{\text{type}} \rightarrow \text{Set}_{\text{type}}^2 \leftarrow \text{Set}_{\text{type}}^2 : 1$ with constant functor $\Delta : \text{Set}_{\text{type}} \rightarrow \text{Set}_{\text{type}}^2$. More specifically, a type span morphism, from source span $x = (X, (X_0, X_1), (x_0, x_1))$ to target span $x' = (X', (X'_0, X'_1), (x'_0, x'_1))$, is a pair $(f, (f_0, f_1))$ consisting of a vertex function $f : X \rightarrow X'$ and a function pair $(f_0, f_1) : (X_0, X_1) \rightarrow (X'_0, X'_1)$, which satisfy the identity $f \cdot \langle x' \rangle = \langle x \rangle \cdot (f_0 \times f_1)$.

However, we do not need full generality here. Instead we restrict the definition by requiring the component functions (f_0, f_1) to be identities. A type span *2-cell* $\alpha : x \xRightarrow{f} x'$, from span $x = (X_0 \xrightarrow{x_0} X \xrightarrow{x_1} X_1)$ to span $x' = (X_0 \xrightarrow{x'_0} X' \xrightarrow{x'_1} X_1)$, is a vertex function $f : X \rightarrow X'$ satisfying the identity $f \cdot \langle x' \rangle = \langle x \rangle$; or equivalently, the identities $f \cdot x'_0 = x_0$ and $f \cdot x'_1 = x_1$ (Figure 10).

```
(iff:set 2-cell)

(iff:function source)
(= (iff:source source) 2-cell)
(= (iff:target source) type.spn:span)

(iff:function target)
(= (iff:source target) 2-cell)
(= (iff:target target) type.spn:span)

(iff:function function) (iff:function vertex) (= vertex function)
(= (iff:source function) 2-cell)
(= (iff:target function) type.ftn:function)

(forall ((2-cell ?a))
  (and (= (type.ftn:composition [(function ?a) (type.spn:function0 (target ?a))])
        (type.spn:function0 (source ?a)))
        (= (type.ftn:composition [(function ?a) (type.spn:function1 (target ?a))])
          (type.spn:function1 (source ?a)))))
```

For convenience of expression, we name the component sets of a type span morphism.

```
(iff:function set0)
(= (iff:source set0) 2-cell)
(= (iff:target set0) type.set:set)
(forall ((2-cell ?a))
```

```

      (and (= (set0 ?a) (type.spn:set0 (source ?a)))
            (= (set0 ?a) (type.spn:set0 (target ?a))))

      (iff:function set1)
      (= (iff:source set1) 2-cell)
      (= (iff:target set1) type.set:set)
      (forall ((2-cell ?a))
        (and (= (set1 ?a) (type.spn:set0 (source ?a)))
              (= (set1 ?a) (type.spn:set0 (target ?a))))))

```

For any two spans $x, x' \in \text{spn}_{\text{type}}$, x belongs to x' , denoted $x \in x'$, when there is a span 2-cell $\alpha : x \xrightarrow{f} x'$ from x to x' . Equivalently, x belongs to x' when the associated functions satisfy belonging: there exists a *proof* function $f \in \text{ftn}_{\text{type}}$ such that $f \cdot \langle x' \rangle = \langle x \rangle$.

$$\begin{array}{ccc}
 & \xrightarrow{f} & \\
 \langle x \rangle & \searrow & \swarrow \langle x' \rangle \\
 & X_0 \times X_1 &
 \end{array}$$

When x' is a span injection (equivalently, $\langle x' \rangle$ is a function injection), the proof function f is unique. We name the component *elements* of each belonging relationship. The belonging relation is a preorder (reflexive and transitive).

```

      (iff:set belonging)
      (forall ((belonging ?xy))
        (exists ((type.spn:span ?x) (type.spn:span ?y))
          (= ?xy [?x ?y])))
      (forall ((type.spn:span ?x) (type.spn:span ?y))
        (<=> (belonging [?x ?y])
              (exists ((2-cell ?a)
                        (and (= (source ?a) ?x)
                             (= (target ?a) ?y))))))

      (forall ((type.spn:span ?x) (type.spn:span ?y))
        (<=> (belonging [?x ?y])
              (type.ftn:belonging [(type.spn:function ?x) (type.spn:function ?y)])))

      (iff:function element0)
      (= (iff:source element0) belonging)
      (= (iff:target element0) type.spn:span)
      (forall ((type.spn:span ?x) (type.spn:span ?y) (belonging [?x ?y]))
        (= (element0 [?x ?y]) ?x))

      (iff:function element1)
      (= (iff:source element1) belonging)
      (= (iff:target element1) type.spn:span)
      (forall ((type.spn:span ?x) (type.spn:span ?y) (belonging [?x ?y]))
        (= (element1 [?x ?y]) ?y))

      (forall ((type.spn:span ?x) (type.spn:injection ?y) (belonging [?x ?y]))
        (forall ((2-cell ?a1) (2-cell ?a2))
          (=> (and (= ?x (source ?a1)) (= ?y (target ?a1))
                  (= ?x (source ?a2)) (= ?y (target ?a2)))
              (= ?a1 ?a2))))

      (forall ((type.spn:span ?x))

```

```

(belonging [?x ?x]))
(forall ((type.spn:span ?x) (type.spn:span ?y) (type.spn:span ?z))
  (=> (and (belonging [?x ?y]) (belonging [?y ?z]))
    (belonging [?x ?z])))

```

Two spans $x, x' \in \text{spn}_{\text{type}}$ are *equivalent*, $x \equiv x'$, when each belongs to the other, $x \sqsubseteq x'$ and $x' \sqsubseteq x$. The equivalence relation is an equivalence relation (reflexive, symmetric and transitive). Two equivalent spans are *isomorphic*, $x \cong x'$, when there exists a bijection $p \in \text{ftn}_{\text{type}}$ proving belonging. The isomorphism relation is an equivalence relation (reflexive, symmetric and transitive).

```

(iff:set equivalence)
(forall ((equivalence ?xy))
  (exists ((type.spn:span ?x) (type.spn:span ?x))
    (= ?xy [?x ?y])))
(forall ((type.spn:span ?x) (type.spn:span ?y))
  (<=> (equivalence [?x ?y])
    (and (belonging [?x ?y]) (belonging [?y ?x]))))

(forall ((type.spn:span ?x))
  (equivalence [?x ?x]))
(forall ((type.spn:span ?x) (type.spn:span ?y))
  (=> (equivalence [?x ?y]) (equivalence [?y ?x])))
(forall ((type.spn:span ?x1) (type.spn:span ?x2) (type.spn:span ?x3))
  (=> (and (equivalence [?x1 ?x2]) (equivalence [?x2 ?x3]))
    (equivalence [?x1 ?x3])))

(iff:set isomorphism)
(forall ((isomorphism ?xy)) (equivalence [?xy]))
(forall ((type.spn:span ?x) (type.spn:span ?y))
  (<=> (isomorphism [?x ?y])
    (exists ((2-cell ?a) (= ?x (source ?a)) (= ?y (target ?a)))
      (type.spn:bijection (function ?a)))))

(forall ((type.spn:span ?x))
  (isomorphism [?x ?x]))
(forall ((type.spn:span ?x) (type.spn:span ?y))
  (=> (isomorphism [?x ?y]) (isomorphism [?y ?x])))
(forall ((type.spn:span ?x1) (type.spn:span ?x2) (type.spn:span ?x3))
  (=> (and (isomorphism [?x1 ?x2]) (isomorphism [?x2 ?x3]))
    (isomorphism [?x1 ?x3])))

```

Conversion. A type span 2-cell $\alpha : x \xrightarrow{f} x'$ has an associated type function 2-cell $\langle \alpha \rangle : \langle x \rangle \xrightarrow{f} \langle x' \rangle$ between the functions of the source and target spans. The function of $\langle \alpha \rangle$, which is the function of α , $f : X \rightarrow X'$, commutes with source and target functions: $f \cdot \langle x' \rangle = \langle x \rangle$.

```

(iff:function function-2-cell)
(= (iff:source function-2-cell) 2-cell)
(= (iff:target function-2-cell) type.ftn.mor:2-cell)
(forall ((2-cell ?a))
  (and (= (type.ftn.mor:source (function-2-cell ?a)) (type.spn:function (source ?a)))
    (= (type.ftn.mor:target (function-2-cell ?a)) (type.spn:function (target ?a)))
    (= (type.ftn.mor:function (function-2-cell ?a)) (function ?a))))

```

Belonging implies inclusion: $x \in x'$ implies $\text{rel}(x) \subseteq \text{rel}(x')$. That is, for any two spans $x, x' \in \text{spn}_{\text{type}}$, if x belongs to x' , then $\text{rel}(x)$ is included in $\text{rel}(x')$.

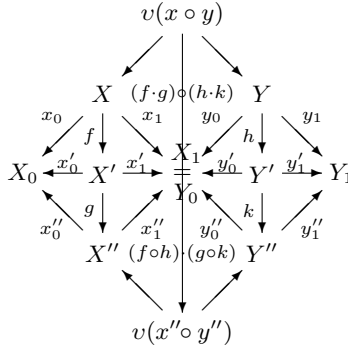
```

(forall ((type.spn:span ?x) (type.spn:span ?y))
  (=> (belonging [?x ?y])
    (type.rel:inclusion-relation [(type.spn:relation ?x) (type.spn:relation ?y)])))

```

Category Theory. Vertical and horizontal compositions satisfy the “interchange” law

$$(\alpha \cdot \beta) \circ (\gamma \cdot \delta) \cong (\alpha \circ \beta) \cdot (\gamma \circ \delta).$$



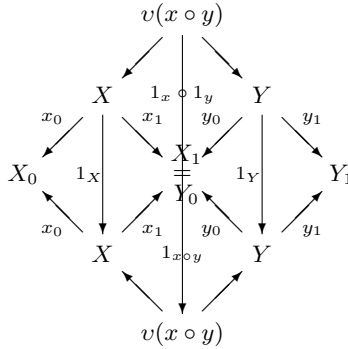
```

(forall ((2-cell ?a) (2-cell ?b) (2-cell ?c) (2-cell ?d)
  (type.spn.mor.vrt:composable-pair [?a ?b]) (type.spn.mor.vrt:composable-pair [?c ?d])
  (type.spn.mor.hrz:composable-pair [?a ?c]) (type.spn.mor.hrz:composable-pair [?b ?d]))
  (type.spn:isomorphism
    [(type.spn.mor.hrz:composition
      [(type.spn.mor.vrt:composition [?a ?b]) (type.spn.mor.vrt:composition [?c ?d])])
      (type.spn.mor.vrt:composition
        [(type.spn.mor.hrz:composition [?a ?c]) (type.spn.mor.hrz:composition [?b ?d])])])])

```

For a composable pair of spans $x = (X_0 \xleftarrow{x_0} X \xrightarrow{x_1} X_1)$ and $y = (Y_0 \xleftarrow{y_0} Y \xrightarrow{y_1} Y_1)$, vertical identity and horizontal composition satisfy the law

$$1_x \circ 1_y = 1_{x \circ y}.$$



```

(forall ((type.spn:span ?x) (type.spn:span ?y) (type.spn:composable-pair [?x ?y]))
  (= (type.spn.mor.hrz:composition [(type.spn.mor.vrt:identity ?x) (type.spn.mor.vrt:identity ?y)])
    (type.spn.mor.vrt:identity (type.spn:composition [?x ?y])))

```

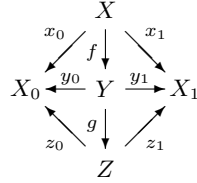
Vertical Aspect. A pair of type span 2-cells is *vertically composable* when the target span of the first is equal to the source span of the second. We name the vertical projection *factors* of vertically composable pairs.

```
(iff:set composable-pair)
(forall ((composable-pair ?ab))
  (exists ((type.spn.mor:2-cell ?a) (type.spn.mor:2-cell ?b))
    (= ?ab [?a ?b])))
(forall ((type.spn.mor:2-cell ?a) (type.spn.mor:2-cell ?b))
  (<=> (composable-pair [?a ?b])
    (= (type.spn.mor:target ?a) (type.spn.mor:source ?b))))

(iff:function vertical-factor0)
(= (iff:source vertical-factor0) composable-pair)
(= (iff:target vertical-factor0) type.spn.mor:2-cell)
(forall ((type.spn.mor:2-cell ?a) (type.spn.mor:2-cell ?b) (composable-pair [?a ?b]))
  (= (vertical-factor0 [?a ?b]) ?a))

(iff:function vertical-factor1)
(= (iff:source vertical-factor1) composable-pair)
(= (iff:target vertical-factor1) type.spn.mor:2-cell)
(forall ((type.spn.mor:2-cell ?a) (type.spn.mor:2-cell ?b) (composable-pair [?a ?b]))
  (= (vertical-factor1 [?a ?b]) ?b))
```

The *vertical composition* of two vertically composable type span 2-cells $\alpha : x \xrightarrow{f} y$ and $\beta : y \xrightarrow{g} z$ is a type span 2-cell $\alpha \cdot \beta : x \xrightarrow{f \cdot g} z$. The vertical source of the composite is the vertical source of the first component factor, the vertical target of the composite is the vertical target of the second component factor, and the function of the composite is the composite of the functions of the component factors.

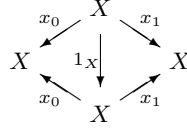


```
(iff:function composition)
(= (iff:source composition) composable-pair)
(= (iff:target composition) type.spn.mor:2-cell)
(forall ((type.spn.mor:2-cell ?a) (type.spn.mor:2-cell ?b) (composable-pair [?a ?b]))
  (and (= (type.spn.mor:source (composition [?a ?b])) (type.spn.mor:source ?a))
    (= (type.spn.mor:target (composition [?a ?b])) (type.spn.mor:target ?b))
    (= (type.spn.mor:function (composition [?a ?b]))
      (type.ftn:composition [(type.spn.mor:function ?a) (type.spn.mor:function ?b)]))))))
```

Composition is associative. Any three vertically composable type span 2-cells $\alpha : x \xrightarrow{f} y$, $\beta : y \xrightarrow{g} z$ and $\gamma : z \xrightarrow{c} w$ satisfy the associative law $\alpha \cdot (\beta \cdot \gamma) = (\alpha \cdot \beta) \cdot \gamma$.

```
(forall ((type.spn.mor:2-cell ?a) (type.spn.mor:2-cell ?b) (type.spn.mor:2-cell ?c)
  (composable-pair [?a ?b]) (composable-pair [?b ?c]))
  (= (composition [?a (composition [?b ?c])])
    (composition [(composition [?a ?b]) ?c])))
```

For every type span $x = (X_0 \xleftarrow{x_0} X \xrightarrow{x_1} X_1)$, there is a unique associated *vertical identity* type span 2-cell $1_x : x \overset{1_x}{\rightrightarrows} x$.



```

(iff:function identity)
(= (iff:source identity) type.spn:span)
(= (iff:target identity) type.spn.mor:2-cell)
(forall ((type.spn:span ?x))
  (and (= (type.spn.mor:source (identity ?x)) ?x)
        (= (type.spn.mor:target (identity ?x)) ?x)
        (= (type.spn.mor:function (identity ?x)) (type.ftn:identity (type.spn:vertex ?x))))))

```

Vertical identity satisfies two unit laws with respect to vertical composition: vertical composition with the vertical identity of the source (target) span of a span 2-cell $\alpha : x \xrightarrow{f} y$ returns that span 2-cell: $1_x \cdot \alpha = \alpha = \alpha \cdot 1_y$.

```

(forall ((type.spn.mor:2-cell ?a))
  (and (= (composition [(identity (type.spn.mor:source ?a)) ?a] ?a)
        (= ?a (composition [?a (identity (type.spn.mor:target ?a))])))

```

Horizontal Aspect. A pair of type span 2-cells is *horizontally composable* when the target set of the first is equal to the source set of the second. We name the horizontal projection *factors* of horizontally composable pairs.

```

(iff:set composable-pair)
(forall ((composable-pair ?ab))
  (exists ((type.spn.mor:2-cell ?a) (type.spn.mor:2-cell ?b))
    (= ?ab [?a ?b])))
(forall ((type.spn.mor:2-cell ?a) (type.spn.mor:2-cell ?b))
  (<=> (composable-pair [?a ?b])
        (= (type.spn.mor:set1 ?a) (type.spn.mor:set0 ?b))))

(iff:function factor0)
(= (iff:source factor0) composable-pair)
(= (iff:target factor0) type.spn.mor:2-cell)
(forall ((type.spn.mor:2-cell ?a) (type.spn.mor:2-cell ?b) (composable-pair [?a ?b]))
  (= (factor0 [?a ?b]) ?a))

(iff:function factor1)
(= (iff:source factor1) composable-pair)
(= (iff:target factor1) type.spn.mor:2-cell)
(forall ((type.spn.mor:2-cell ?a) (type.spn.mor:2-cell ?b) (composable-pair [?a ?b]))
  (= (factor1 [?a ?b]) ?b))

```

For any horizontally composable pair of span 2-cells $\alpha : x \xrightarrow{f} x'$ and $\beta : y \xrightarrow{g} y'$, the pair of source (target) spans is composable.

```

(forall ((type.spn.mor:2-cell ?a) (type.spn.mor:2-cell ?b) (composable ?a ?b))
  (and (type.spn:composable (type.spn.mor:source ?a) (type.spn.mor:source ?b))
        (type.spn:composable (type.spn.mor:target ?a) (type.spn.mor:target ?b))))

```

For any horizontally composable pair of span 2-cells $\alpha : x \xrightarrow{f} x'$ and $\beta : y \xrightarrow{g} y'$, there is an associated opspan morphism

$$\Upsilon(\alpha, \beta) \left\{ \begin{array}{c} \overbrace{\begin{array}{ccc} v(x) & \xrightarrow{\varphi_1^x} & \sigma_1(x) = \sigma_0(y) & \xleftarrow{\varphi_0^y} & v(y) \\ f \downarrow & & 1_{X_1} \neq 1_{Y_0} & & \downarrow g \\ v(x') & \xrightarrow{\varphi_1^{x'}} & \sigma_1(x') = \sigma_0(y') & \xleftarrow{\varphi_0^{y'}} & v(y') \end{array}}^{\Upsilon(x, y)} \\ \underbrace{\hspace{10em}}_{\Upsilon(x', y')} \end{array} \right.$$

```

(iff:function opspan-morphism)
(= (iff:source opspan-morphism) composable-pair)
(= (iff:target opspan-morphism) type.dgm.ospn.mor:opspan-morphism)
(forall ((type.spn.mor:2-cell ?a) (type.spn.mor:2-cell ?b) (composable-pair [?a ?b]))
  (and (= (type.dgm.ospn.mor:source (opspan-morphism [?a ?b]))
    (type.spn:opspan [(type.spn.mor:source ?a) (type.spn.mor:source ?b)]))
    (= (type.dgm.ospn.mor:target (opspan-morphism [?a ?b]))
    (type.spn:opspan [(type.spn.mor:target ?a) (type.spn.mor:target ?b)]))
    (= (type.dgm.ospn.mor:function0 (opspan-morphism [?a ?b])) (type.spn.mor:function ?a))
    (= (type.dgm.ospn.mor:function1 (opspan-morphism [?a ?b])) (type.spn.mor:function ?b))))
(= (type.dgm.ospn.mor:opvertex (opspan-morphism [?a ?b]))
  (type.ftn:identity (type.spn.mor:set1 ?a)))
(= (type.dgm.ospn.mor:opvertex (opspan-morphism [?a ?b]))
  (type.ftn:identity (type.spn.mor:set0 ?b)))

```

The *horizontal composition* of two horizontally composable type span 2-cells $\alpha : x \xrightarrow{f} x'$ and $\beta : y \xrightarrow{g} y'$ is a type span 2-cell $\alpha \circ \beta : x \circ y \xrightarrow{\lim \Upsilon(\alpha, \beta)} x' \circ y'$. The horizontal source of the composite is the composite of the source spans of the component factors, the horizontal target of the composite is the composite of the target spans of the component factors, and the function of the composite is the pullback of the opspan morphism of the horizontally composable pair.

$$\begin{array}{ccccc} & & v(x \circ y) = X \times_{X_1=Y_0} Y & & \\ & \swarrow_{y_0} & \downarrow f \circ g & \searrow_{y_1} & \\ & X & X_1 = Y_0 & Y & \\ & \swarrow_{x_0} & \downarrow f & \downarrow g & \searrow_{y_1} \\ X_0 & & X_1 & & Y_1 \\ & \swarrow_{x'_0} & \downarrow f & \downarrow g & \searrow_{y'_1} \\ & X' & X_1 = Y_0 & Y' & \\ & \swarrow_{y'_0} & \downarrow f & \downarrow g & \searrow_{y'_1} \\ & & v(x' \circ y') = X' \times_{X'_1=Y'_0} Y' & & \end{array}$$

```

(iff:function composition)
(= (iff:source composition) composable-pair)
(= (iff:target composition) type.spn.mor:2-cell)
(forall ((type.spn.mor:2-cell ?a) (type.spn.mor:2-cell ?b) (composable-pair [?a ?b]))
  (and (= (type.spn.mor:source (composition [?a ?b]))
    (type.spn:composition [(type.spn.mor:source ?a) (type.spn.mor:source ?b)]))
    (= (type.spn.mor:target (composition [?a ?b]))
    (type.spn:composition [(type.spn.mor:target ?a) (type.spn.mor:target ?b)]))
    (= (type.spn.mor:function (composition [?a ?b]))
    (type.lim.pbk.mor:pullback (opspan-morphism [?a ?b])))))

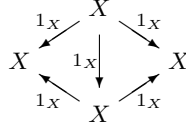
```

Composition is associative (up to natural isomorphism)¹⁰. Any three horizontally composable type span 2-cells $\alpha : x \xrightarrow{f} x'$, $\beta : y \xrightarrow{g} y'$ and $\gamma : z \xrightarrow{h} z'$ satisfy the associative law

$$\alpha \circ (\beta \circ \gamma) \cong (\alpha \circ \beta) \circ \gamma.$$

```
(forall ((type.spn.mor:2-cell ?a) (type.spn.mor:2-cell ?b) (type.spn.mor:2-cell ?c)
  (composable-pair [?a ?b]) (composable-pair [?b ?c]))
  (type.spn.mor:isomorphism
    [(composition [?a (composition [?b ?c])])
     (composition [(composition [?a ?b]) ?c])]))
```

For every type set X , there is a unique associated *horizontal identity* type span 2-cell $1_{1_X} : 1_X \xrightarrow{1_X} 1_X$. The horizontal source of the identity is the identity span $1_X = (X \xleftarrow{1_X} X \xrightarrow{1_X} X)$, the horizontal target of the identity is the identity span $1_X = (X \xleftarrow{1_X} X \xrightarrow{1_X} X)$, and the function of the identity is the identity function $1_X : X \rightarrow X$.



```
(iff:function identity)
(= (iff:source identity) type.set:set)
(= (iff:target identity) type.spn.mor:2-cell)
(forall ((type.set:set ?X))
  (and (= (type.spn.mor:source (identity ?X)) (type.spn:identity ?X))
        (= (type.spn.mor:target (identity ?X)) (type.spn:identity ?X))
        (= (type.spn.mor:function (identity ?x)) (type.ftn:identity ?X))))
```

Horizontal identity satisfies two unit laws with respect to horizontal composition: horizontal composition with the horizontal identity of the source (target) set of a span 2-cell $\alpha : x \xrightarrow{f} x'$, where $x = (X_0 \xleftarrow{x_0} X \xrightarrow{x_1} X_1)$ and $x' = (X_0 \xleftarrow{x'_0} X' \xrightarrow{x'_1} X_1)$, returns that span 2-cell (up to natural isomorphism):

$$1_{1_{X_0}} \circ \alpha \cong \alpha \cong \alpha \circ 1_{1_{X_1}}.$$

```
(forall ((type.spn.mor:2-cell ?a))
  (and (type.spn.mor:isomorphism
        [(composition [(identity (type.spn.mor:set0 ?a)) ?a]) ?a])
        (type.spn.mor:isomorphism
          [?a (composition [?a (identity (type.spn.mor:set1 ?a))])]))))
```

¹⁰The category Set_{type} gives rise to a span bicategory with 0-cells being type sets, 1-cells being type spans and 2-cells being type span 2-cells.

ordinary elements $x \in X$		generalized elements $Y \xrightarrow{x} X$
predicates as inclusions $ p \subseteq X$	predicates as injections $ p \xrightarrow{p} X$	
predicate membership in terms of		
set membership	function application	function composition
$x \in p $	$p(x)$ when $\exists y \in p p(y) = x$	$\exists y : Y \rightarrow p y \cdot p = x$

Table 7: Predicates, Elements and Membership

1.5 Type Predicates

Introduction. Just as sets represent the nouns in natural language expressions, so also predicates represent the adjectives. In the expression “Michael is a young person”, the adjective “young” modifies the noun “person”. The semantics of this expression consists of a set of persons, a subset consisting of young people and an individual asserted to be a member of that subset. Predicates, elements and predicate membership have several representations in mathematics and knowledge engineering (Table 7). Predicates can be represented as subsets (inclusions) or injective functions, elements can be represented as ordinary (global) elements or generalized elements (functions), and the representation of predicate membership varies accordingly. The IFF symbolism for predicate membership, both for ordinary elements and for generalized elements, is defined in this namespace¹¹.

Basics. There is a collection of all type *predicates*. Predicates are also called *parts*. The symbol ‘`predicate`’ is used to declare a type predicate. Conceptually, a type predicate is an injective type function, hence an injective IFF function. But practically, in order to parse the defined syntactic construct of predicate membership, we require the collection of type predicates to be disjoint from the collections of type sets and type functions. The collection of all type predicates is an IFF set. A type predicate can be neither a type set nor a type function. These three collections are pairwise disjoint. The collections of type sets and type functions already inherit their disjointness from the collection of IFF sets and IFF functions.

```
(iff:set predicate)
(forall ((predicate ?p))
  (and (not (type.set:set ?p))
       (not (type.ftn:function ?p))))
```

¹¹Given any type predicate $p : X$, the IFF symbolism for predicate membership is `(p x)` for ordinary elements $x \in X$ and `(member x p)` for generalized elements (functions $x : Y \rightarrow X$).

Each type predicate $p : |p| \hookrightarrow X$ has a unique *differentia* type set $\delta(p) = \text{diff}(p) = |p|$ and a unique *genus* type set $\gamma(p) = \text{gen}(p) = X$. The predicate notation $p : X$ shows a predicate p with genus X .

```
(iff:function differentia)
(= (iff:source differentia) predicate)
(= (iff:target differentia) type.set:set)
```

```
(iff:function genus)
(= (iff:source genus) predicate)
(= (iff:target genus) type.set:set)
```

A predicate with genus X generalizes a subset of X . We can recapture subsets with strictness. A predicate is *strict* when the differentia is a subset of the genus. For strict predicates, the associated injective function is an inclusion (of the differentia into the genus).

```
(iff:set strict-predicate)
(forall ((strict-predicate ?p)) (predicate ?p))
(forall ((predicate ?p))
  (<=> (strict-predicate ?p)
    (type.set:subset (differentia ?p) (genus ?p))))
(forall ((strict-predicate ?p))
  (= (function ?p) (type.set:inclusion [(differentia ?p) (genus ?p)])))
```

For each type predicate $p : |p| \hookrightarrow X$ there is a *canonically* strict predicate $[p] : \rho(p) \subseteq X$ of the same genus, whose differentia is the range of the injective function of p .

```
(iff:function canon)
(= (iff:source canon) predicate)
(= (iff:target canon) predicate)
(forall ((predicate ?p))
  (= (differentia (canon ?p)) (type.ftn:range (function ?p)))
  (= (genus (canon ?p)) (genus ?p))
  (= (function (canon ?p)) (type.ftn:injective-factor (function ?p))))
```

There is a binary *delimitation* relationship \leq between pairs of type predicates. One (*smaller*) type predicate $|p| \xrightarrow{p} X$ is the delimitation of another (*larger*) type predicate $|q| \xrightarrow{q} Y$, denoted $p \leq q$, when (1) the genus of p is a subset of the genus of q , $X \hookrightarrow Y$, (2) the differentia of p is a subset of the differentia of q , $|p| \hookrightarrow |q|$, and (3) the function of the predicate p is the optimal restriction of the function of the predicate q . When both predicates are strict, p is the delimitation of q *iff* for all $x \in X$, $p(x)$ iff $q(x)$. The delimitation endorelation is a partial order (reflexive, antisymmetric and transitive).

$$\begin{array}{ccc}
 |p| & \xhookrightarrow{p} & X \\
 \downarrow & & \downarrow \\
 |q| & \xhookrightarrow{q} & Y
 \end{array}
 \quad \lrcorner$$

```
(iff:set delimitation-relation)
(forall ((delimitation-relation ?pq))
  (exists ((predicate ?p) (predicate ?q))
```

For each type predicate $p : X$, the differentia embeds as the subset of the genus $\text{diff}(p) \hookrightarrow \text{gen}(p)$, consisting of those (ordinary) elements $x \in X$ satisfying the predicate: $p(x)$ iff $\exists_{y \in \text{diff}(p)} p(y) = x$. Hence, we introduce into the IFF the predicate holds statement ‘(p x)’. If the symbols ‘p’ and ‘X’ represent the two IFF things p and X , then the code

```
(predicate p)
(set X) (= (genus p) X)
(set Y) (= (differentia p) Y)
```

makes the declaration “ $p : Y \hookrightarrow X$ ”^a, and the code

```
(X x)
```

expresses the statement that “ $x \in X$ ”. All of this follows standard IFF syntax, which, until now, was expressed in terms of set membership and function application. However, the following code

```
(p x)
```

is new. Here the symbol ‘p’ is neither a set nor a function, and hence we can use neither set membership nor function application to define this. The predicate holds expression ‘(p x)’, which states that ‘x’ satisfies ‘p’, serves as a shorthand for the code

```
(exists ((Y ?y))
  (= ((function p) ?y) x))
```

This follows standard IFF syntax, since it is expressed only in terms of set membership and function application. Hence, the following equivalence holds anywhere in the IFF

```
(forall ((predicate ?p) ((genus ?p) ?x))
  (<=> (?p ?x)
    (exists (((differentia ?p) ?y))
      (= ((function ?p) ?y) ?x))))
```

The notation ‘(p x)’ follows the prescription: *all IFF predicates are unary*. Hence, the following nullary, binary, ternary and higher arity expressions are iff-formed

```
(p)
(p x0 x1)
(p x0 x1 x2)
...
```

^aequivalently, “ $p \in \text{pred}, \gamma(p) = X, \delta(p) = Y$ ”

Table 8: IFF Predicate Notation

```

      (= (?pq [?p ?q])))
    (forall ((predicate ?p) (predicate ?q))
      (<=> (delimitation-relation [?p ?q])
        (and (type.set:subset-relation [(genus ?p) (genus ?q)])
              (type.set:subset-relation [(differentia ?p) (differentia ?q)])
              (type.ftn:optimal-restriction-relation [(function ?p) (function ?q)]))))

    (iff:function smaller)
    (= (iff:source smaller) delimitation-relation)
    (= (iff:target smaller) predicate)
    (forall ((predicate ?p) (predicate ?q) (delimitation-relation [?p ?q]))
      (= (smaller [?p ?q] ?p))

    (iff:function larger)
    (= (iff:source larger) delimitation-relation)
    (= (iff:target larger) predicate)
    (forall ((predicate ?p) (predicate ?q) (delimitation-relation [?p ?q]))
      (= (larger [?p ?q] ?q))

    (forall ((strict-predicate ?p) (strict-predicate ?q)
              (type.set:subset-relation [(genus ?p) (genus ?q)]))
      (<=> (delimitation-relation [?p ?q])
        (forall ((genus ?p) ?x)
          (<=> (?p ?x) (?q ?x)))))

    (forall ((predicate ?p))
      (delimitation-relation [?p ?p]))
    (forall ((predicate ?p) (predicate ?q))
      (=> (and (delimitation-relation [?p ?q]) (delimitation-relation [?q ?p]))
        (= ?p ?q))
    (forall ((predicate ?p1) (predicate ?p2) (predicate ?p3))
      (=> (and (delimitation-relation [?p1 ?p2]) (delimitation-relation [?p2 ?p3]))
        (delimitation-relation [?p1 ?p3])))

```

Conversion. A part (predicate) of a set X has an associated type injective *function* (more generally, a monomorphism) $\varphi(p) : |p| \hookrightarrow X$. The source (target) of the injective function is the differentia (genus) of the predicate p .

```

(iff:function function)
(= (iff:source function) predicate)
(= (iff:target function) type.ftn:function)
(forall ((predicate ?p))
  (and (= (type.ftn:source (function ?p)) (differentia ?p))
        (= (type.ftn:target (function ?p)) (genus ?p))
        (type.ftn:injection (function ?p))))

(forall ((predicate ?p1) (predicate ?p2))
  (=> (= (function ?p1) (function ?p2)) (= ?p1 ?p2)))

```

The canon of any predicate is equal to the predicate of its injective function, $[p] = \pi(\varphi(p))$.

```

(forall ((predicate ?p))
  (= (canon ?p) (type.ftn:predicate (function ?p))))

```

Any predicate $p : Y \hookrightarrow X$ has an associated *subordinate* pair $\sigma(p) = (\rho(p), X)$, where $\rho(p) \cong Y$ is the range of the injective function p . This is part of the

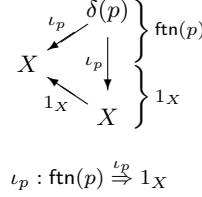


Figure 11: Injection of a Predicate

fact that the canonical restriction of predicates on genus X is isomorphic to the powerset of X .

```
(iff:function subordinate)
(= (iff:source subordinate) predicate)
(= (iff:target subordinate) type.set:subset-relation)
(forall ((predicate ?p))
  (and (= (type.set:smaller (subordinate ?p)) (type.ftn:range (function ?p)))
        (= (type.set:larger (subordinate ?p)) (genus ?p))
        (= (type.set:inclusion (subordinate ?p))
            (type.ftn:injective-factor (function ?p)))))
```

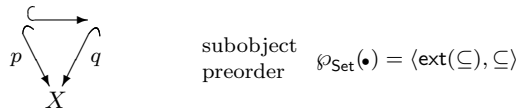
The canon of any predicate is equal to the predicate of its subordinate pair, $[p] = \pi(\sigma(p))$.

```
(forall ((predicate ?p))
  (= (canon ?p) (type.set:predicate (subordinate ?p))))
```

For any predicate $p : X$, there is an *injection* function 2-cell $\iota_p : \text{ftn}(p) \Rightarrow 1_X$ (Figure 11), whose source $\text{ftn}(p)$ is the function (injection) of a predicate, whose target is the terminal (predicative) function 1_X , and whose function $\iota_p : \delta(p) \hookrightarrow X$ is the injection of p .

```
(iff:function injection)
(= (iff:source injection) predicate)
(= (iff:target injection) type.ftn.mor:2-cell)
(forall ((predicate ?p))
  (and (= (type.ftn.mor:source (injection ?p)) (function ?p))
        (= (type.ftn.mor:target (injection ?p)) (type.ftn:terminal (genus ?p)))
        (= (type.ftn.mor:function (injection ?p)) (function ?p))))
```

Subobject. For any two type predicates $p : X$ and $q : Y$, p is included in q , denoted by $p \subseteq q$, when (the function of) p belongs to (the function of) q . When p is included in q , the genus of p is the genus of q , $X = Y$. The inclusion endorelation on predicates is a preorder (reflexive and transitive). We name the component *parts* of an inclusion relationship; that is, there are projections $\pi_0^{\subseteq}, \pi_1^{\subseteq} : \text{ext}(\subseteq) \rightarrow \text{pred}$ to the component parts.



```

(iff:set inclusion-relation)
(forall ((inclusion-relation ?pq))
  (exists ((predicate ?p) (predicate ?q))
    (= ?pq [?p ?q])))
(forall ((predicate ?p) (predicate ?q))
  (<=> (inclusion-relation [?p ?q])
    (type.ftn:belonging [(function ?p) (function ?q)])))

(forall ((predicate ?p) (predicate ?q))
  (=> (inclusion-relation [?p ?q])
    (= (genus ?p) (genus ?q))))

(forall ((predicate ?p))
  (inclusion-relation [?p ?p]))
(forall ((predicate ?p) (predicate ?q) (predicate ?r))
  (=> (and (inclusion-relation [?p ?q]) (inclusion-relation [?q ?r]))
    (inclusion-relation [?p ?r])))

(iff:function part0)
(= (iff:source part0) inclusion-relation)
(= (iff:target part0) predicate)
(forall ((predicate ?p) (predicate ?q) (inclusion-relation [?p ?q]))
  (= (part0 [?p ?q]) ?p))

(iff:function part1)
(= (iff:source part1) inclusion-relation)
(= (iff:target part1) predicate)
(forall ((predicate ?p) (predicate ?q) (inclusion-relation [?p ?q]))
  (= (part1 [?p ?q]) ?q))

```

The inclusion relation on predicates generalizes the inclusion relation on the powerset of X . We can recapture the latter through the canonically strict associate: for any two type predicates p and q , $p \subseteq q$ iff $[p] \subseteq [q]$.

```

(forall ((predicate ?p) (predicate ?q))
  (<=> (inclusion-relation [?p ?q])
    (inclusion-relation [(canon ?p) (canon ?q)])))

```

For any two type predicates p and q , p is *equivalent* or *isomorphic* to q , denoted by $p \equiv q$, when p and q are included in each other, $p \subseteq q$ and $q \subseteq p$. When p is equivalent to q , the differentia of p is isomorphic to the differentia of q , $|p| \cong |q|$, and p and q factor through each other via the inverses. For any two type predicates p and q , p is equivalent to q iff the canonically strict associates are equal: $p \equiv q$ iff $[p] = [q]$. The equivalence endorelation on type predicates is an equivalence relation (reflexive, symmetric and transitive). Note: for any type predicate p , the collection of type predicates equivalent to p , the equivalence class $[p]$, is not necessarily locally small.

```

(iff:set equivalence) (iff:set isomorphism) (= isomorphism equivalence)
(forall ((equivalence ?pq))
  (exists ((predicate ?p) (predicate ?q))
    (= ?pq [?p ?q])))
(forall ((predicate ?p) (predicate ?q))
  (<=> (equivalence [?p ?q])
    (and (inclusion-relation [?p ?q])
      (inclusion-relation [?q ?p]))))
(forall ((predicate ?p) (predicate ?q))
  (<=> (equivalence [?p ?q])
    (and (inclusion-relation [?p ?q])
      (inclusion-relation [?q ?p]))))

```

```

(= (canon ?p) (canon ?q)))

(forall ((predicate ?p))
  (equivalence [?p ?p]))
(forall ((predicate ?p) (predicate ?q))
  (=> (equivalence [?p ?q]) (equivalence [?q ?p])))
(forall ((predicate ?p) (predicate ?q) (predicate ?r))
  (=> (and (equivalence [?p ?q]) (equivalence [?q ?r]))
      (equivalence [?p ?r])))

```

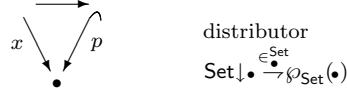
Any predicate is equivalent to its canonically strict associate, $p \equiv [p]$. Hence, for any type predicate p , the equivalence class of p has the canon as canonical representative.

```

(forall ((predicate ?p))
  (equivalence [?p (canon ?p)]))

```

For any type function (generalized element) $x \xrightarrow{\bullet} \bullet$ and any type predicate (part) $\xrightarrow{p} \bullet$, x is a *member* of p , $x \in p$, when x belongs to (the function of) p ; that is, when there is a proof function h such that $x = h \cdot p$. As noted when defining the belonging relation, the proof function h is unique. The membership relation from type functions to type predicates is closed with respect to function belonging on the left and predicate inclusion on the right. We name the component *function* and *part* of a membership relationship; that is, there are projections $\pi_0^\in \text{ : ext}(\in) \rightarrow \text{ftn}$ and $\pi_1^\in \text{ : ext}(\in) \rightarrow \text{pred}$ to the components.



```

(iff:set membership)
(forall ((membership ?xp))
  (exists ((type.ftn:function ?x) (predicate ?p))
    (= ?xp [?x ?p])))
(forall ((type.ftn:function ?x) (predicate ?p))
  (<=> (membership [?x ?p])
      (type.ftn:belonging [?x (function ?p)])))

(forall ((type.ftn:function ?x) (predicate ?p) (membership [?x ?p]))
  (and (forall ((type.ftn:function ?y) (type.ftn:belonging [?y ?x]))
        (membership [?y ?p]))
      (forall ((predicate ?q) (inclusion-relation [?p ?q]))
        (membership [?x ?q]))))

(iff:function element)
(= (iff:source element) membership)
(= (iff:target element) type.ftn:function)
(forall ((type.ftn:function ?x) (predicate ?p) (membership [?x ?p]))
  (= (element [?x ?p]) ?x))

(iff:function part)
(= (iff:source part) membership)
(= (iff:target part) predicate)
(forall ((type.ftn:function ?x) (predicate ?p) (membership [?x ?p]))
  (= (part [?x ?p]) ?p))

```

Fact 2 *Inclusion is equivalent to universal implication of membership*

$$p \subseteq q \text{ iff } \forall_{x \in X} (x \in p \Rightarrow x \in q) \quad \subseteq = \in \setminus \in$$

Hence, the usual relationship holds between inclusion and membership.

```
(forall ((predicate ?p) (predicate ?q))
  (<=> (inclusion-relation [?p ?q])
    (forall ((type.ftn:function ?x))
      (=> (membership [?x ?p]) (membership [?x ?q])))))
```

We name the unique *proof* function for membership.

```
(iff:function proof)
(= (iff:source proof) membership)
(= (iff:target proof) type.ftn:function)
(forall ((type.ftn:function ?x) (predicate ?p) (membership [?x ?p]))
  (and (= (type.ftn:source (proof [?x ?p])) (type.ftn:source ?x))
    (= (type.ftn:target (proof [?x ?p])) (differentia ?p))
    (= ?x (type.ftn:composition [(proof [?x ?p]) (function ?p)]))))
```

ordinary element pairs $(x_0, x_1) \in X_0 \times X_1$		generalized element pairing $Y \xrightarrow{\langle x_0, x_1 \rangle} X_0 \times X_1$
relations as inclusions $\text{ext}(r) \subseteq X_0 \times X_1$	relations as injections $\text{ext}(r) \xrightarrow{r} X_0 \times X_1$	
relation membership in terms of		
set membership	function application	function composition
$r(x_0, x_1)$ when		
$(x_0, x_1) \in \text{ext}(r)$	$\exists y \in \text{ext}(r) r(y) = (x_0, x_1)$	$\exists y. Y \rightarrow \text{ext}(r) y \cdot r = \langle x_0, x_1 \rangle$

Table 9: Relations, Elements and Membership

1.6 Type Relations

Introduction. Just as sets represent the nouns in natural language expressions, and predicates represent the adjectives, so also relations represent the verbs. In the expression “Elizabeth marries Darcy”, the verb “marry” links the subject “Elizabeth” to the direct object “Darcy”. The semantics of this expression consists of a pair of individuals asserted to be a member of a binary relation. In mathematics and knowledge engineering, relations, element pairs and relation membership have several representations, as indicated in Table 9. Relations can be represented as subsets (inclusions) or injections, element pairs can be represented as ordinary (global) element pairs or generalized element (function) pairing, and the representation of relation membership varies accordingly¹².

Basics. There is a collection of all type *relations*. The symbol ‘`relation`’ is used to declare a type relation. Conceptually, a type relation is a type predicate, hence an injective type function. But practically, in order to parse the defined syntactic construct of relation membership, we require the collection of type relations to be disjoint from the collections of type sets, type functions and type predicates. The collection of all type relations is an IFF set. A type relation can be neither a type set, a type function nor a type predicate. These four collections are pairwise disjoint. The collections of type sets and type functions already inherit their disjointness from the collection of IFF sets and IFF functions. The disjointness of the collection of IFF predicates was axiomatized before.

```
(iff:set relation)
(forall ((relation ?r))
  (and (not (type.set:set ?r))
       (not (type.ftn:function ?r))))
(not (type.pred:predicate ?r))))
```

¹²Given any type relation $r : X_0 \rightarrow X_1$, the IFF symbolism for relation membership is (`r x0 x1`) for ordinary element pairs (x_0, x_1) and (`member x r`) for generalized element pairs (spans $x = (x_0 : X_0 \leftarrow Y \rightarrow X_1 : x_1)$).

Each type relation $r : \text{ext}(r) \hookrightarrow X_0 \times X_1$ has a unique *extent* type set $\varepsilon(r) = \text{ext}(r)$ and a unique *component* type set pair $\sigma(r) = \text{set}(r) = (X_0, X_1)$. The relation notation $r : X_0 \rightarrow X_1$ shows a relation r with domain or zeroth component type set X_0 and codomain or first component type set X_1 . The notation $\sigma_0(r) = X_0$ and $\sigma_1(r) = X_1$ is also used to assert the domain (zeroth component) and codomain (first component). For convenience we name the domain-codomain pairing map for type relations.

```
(iff:function extent)
(= (iff:source extent) relation)
(= (iff:target extent) type.set:set)

(iff:function set0)
(= (iff:source set0) relation)
(= (iff:target set0) type.set:set)

(iff:function set1)
(= (iff:source set1) relation)
(= (iff:target set1) type.set:set)

(iff:function set-pair)
(= (iff:source set-pair) relation)
(= (iff:target set-pair) type.dgm.pr.obj:set-pair)
(forall ((relation ?r))
  (= (set-pair ?r) [(set0 ?r) (set1 ?r)]))
```

For any type relation, there are two *projection* type functions $\pi_0^r = \text{ftn}(r) \cdot \pi_0 : \text{ext}(r) \rightarrow X_0$ and $\pi_1^r = \text{ftn}(r) \cdot \pi_1 : \text{ext}(r) \rightarrow X_1$.

```
(iff:function projection0)
(= (iff:source projection0) relation)
(= (iff:target projection0) type.ftn:function)
(forall ((relation ?r))
  (and (= (type.ftn:source (projection0 ?r)) (extent ?r))
        (= (type.ftn:target (projection0 ?r)) (set0 ?r))
        (= (projection0 ?r)
            (type.ftn:composition
              [(function ?r) (type.lim.prd2.obj:projection0 (set-pair ?r)]))))))

(iff:function projection1)
(= (iff:source projection1) relation)
(= (iff:target projection1) type.ftn:function)
(forall ((relation ?r))
  (and (= (type.ftn:source (projection1 ?r)) (extent ?r))
        (= (type.ftn:target (projection1 ?r)) (set0 ?r))
        (= (projection1 ?r)
            (type.ftn:composition
              [(function ?r) (type.lim.prd2.obj:projection1 (set-pair ?r)]))))))
```

A relation $r : X_0 \rightarrow X_1$ generalizes a subset of $X_0 \times X_1$. We can recapture subsets with strictness. A relation is *strict* when the extent is a subset of the product of the associated set pair. For strict relations, the associated injective function is an inclusion (of the extent into this product). A relation is strict iff its associated predicate is strict.

```
(iff:set strict-relation)
(forall ((strict-relation ?r)) (relation ?r))
(forall ((relation ?r))
```

For each type relation $r : X_0 \rightarrow X_1$, the extent embeds as the subset of the product of the component set pair $\text{ext}(r) \hookrightarrow \text{set}_0(r) \times \text{set}_1(r)$, consisting of those (ordinary) element pairs $(x_0, x_1) \in X_0 \times X_1$ satisfying the relation: $r(x_0, x_1)$ iff $\exists_{y \in \text{ext}(r)} r(y) = (x_0, x_1)$. Hence, we introduce into the IFF the relational holds statement ' $(r \ x0 \ x1)$ '. If the symbols ' r ', ' $X0$ ' and ' $X1$ ' represent the three IFF things r , X_0 and X_1 , then the code

```
(relation r)
(set X0) (= (set0 r) X0)
(set X1) (= (set1 r) X1)
(set Y) (= (extent r) Y)
```

makes the declaration " $r : Y \hookrightarrow X_0 \times X_1$ "^a, and the code

```
(X0 x0) (X1 x1)
```

expresses the statement that " $(x_0, x_1) \in X_0 \times X_1$ ". All of this follows standard IFF syntax, which, until now, was expressed in terms of set membership and function application. However, the following code

```
(r x0 x1)
```

is new. Here the symbol ' r ' is neither a set, a function nor a predicate, and hence we can use neither set membership, function application nor predicate holds to define this. The relation holds expression ' $(r \ x0 \ x1)$ ', which states that ' $(x0 \ x1)$ ' satisfies ' r ', serves as a shorthand for the code

```
(exists ((Y ?y))
 (= ((function r) ?y) [x0 x1]))
```

This follows standard IFF syntax, since it is expressed only in terms of set membership and function application. Hence, the following equivalence holds anywhere in the IFF

```
(forall ((relation ?r) ((set0 ?r) ?x0) ((set1 ?r) ?x1))
 (<=> (?r ?x0 ?x1)
 (exists (((extent ?r) ?y))
 (= ((function ?r) ?y) [?x0 ?x1])))
```

This equivalence can be expressed in terms of predicates

```
(forall ((relation ?r) ((set0 ?r) ?x0) ((set1 ?r) ?x1))
 (<=> (?r ?x0 ?x1)
 ((predicate ?r) [x0 x1]))
```

The notation ' $(r \ x0 \ x1)$ ' follows the prescription: *all IFF relations are binary*. Hence, the following nullary, unary, ternary and higher arity expressions are ill-formed

```
(r)
(r x0)
(r x0 x1 x2)
...
```

^aequivalently, " $r \in \text{rel}, \sigma_0(r) = X_0, \sigma_1(r) = X_1, \varepsilon(r) = Y$ "

Table 10: IFF Relational Notation

```

(<=> (strict-relation ?r)
      (type.set:subset (extent ?r) (type.lim.prd2.obj:product (set-pair ?r))))
(forall ((strict-relation ?r))
  (= (function ?r)
      (type.set:inclusion [(extent ?r) (type.lim.prd2.obj:product (set-pair ?r))])))

(forall ((relation ?r))
  (<=> (strict-relation ?r)
        (type.pred:strict-predicate (predicate ?r))))

```

For each type relation $r : X_0 \rightarrow X_1$, there is a *canonically* strict relation $[r] : X_0 \rightarrow X_1$, whose extent is the range of the injective function of r .

```

(iff:function canon)
(= (iff:source canon) relation)
(= (iff:target canon) relation)
(forall ((relation ?r))
  (= (extent (canon ?r)) (type.ftn:range (function ?r)))
     (= (set-pair (canon ?r)) (set-pair ?r))
     (= (function (canon ?r)) (type.ftn:injective-factor (function ?r))))

```

There is a binary *abridgment* relationship \preceq between pairs of type relations. One (*smaller*) type relation $r : X_0 \rightarrow X_1$ is the abridgment of another (*larger*) type relation $s : Y_0 \rightarrow Y_1$, $r \preceq s$, when (1) the domain (codomain) of r is a subset of the domain (codomain) of s , $X_0 \subseteq Y_0$ ($Y_0 \subseteq Y_1$) (hence, the set-pair product of r is a subset of the set-pair product of s , $X_0 \times X_1 \subseteq Y_0 \times Y_1$), (2) the extent of r is a subset of the extent of s , $\text{ext}(r) \subseteq \text{ext}(s)$, and (3) the function of the relation r is the optimal restriction of the function of the relation s . When both relations are strict, r is the abridgment of s *iff* for all $x_0 \in X_0, x_1 \in X_1$, $r(x_0, x_1)$ *iff* $s(x_0, x_1)$. The abridgment relation is a partial order (reflexive, antisymmetric and transitive).

$$\begin{array}{ccc}
 \text{ext}(r) \hookrightarrow & \xrightarrow{r} & X_0 \times X_1 \\
 \downarrow & & \downarrow \\
 \text{ext}(s) \hookrightarrow & \xrightarrow{s} & Y_0 \times Y_1 \\
 & \lrcorner &
 \end{array}$$

```

(iff:set abridgment-relation)
(forall ((abridgment-relation ?rs))
  (exists ((relation ?r) (relation ?s))
    (= (?rs [?r ?s]))))
(forall ((relation ?r) (relation ?s))
  (<=> (abridgment-relation [?r ?s])
        (and (type.set:subset-relation [(set0 ?r) (set0 ?s)])
              (type.set:subset-relation [(set1 ?r) (set1 ?s)])
              (type.set:subset-relation [(extent ?r) (extent ?s)])
              (type.ftn:optimal-restriction-relation [(function ?r) (function ?s)]))))

```

```

(iff:function smaller)
(= (iff:source smaller) abridgment-relation)
(= (iff:target smaller) relation)
(forall ((relation ?r) (relation ?s) (abridgment-relation [?r ?s]))
  (= (smaller [?r ?s]) ?r))

```

```

(iff:function larger)
(= (iff:source larger) abridgment-relation)
(= (iff:target larger) relation)
(forall ((relation ?r) (relation ?s) (abridgment-relation [?r ?s]))
  (= (larger [?r ?s]) ?s))

(forall ((strict-relation ?r) (strict-relation ?s)
  (type.set:subset-relation [(set0 ?r) (set0 ?s)])
  (type.set:subset-relation [(set1 ?r) (set1 ?s)]))
  (<=> (abridgment-relation [?r ?s])
    (forall ((set0 ?r) ?x0) ((set1 ?r) ?x1)
      (<=> (?r ?x0 ?x1) (?s ?x0 ?x1))))))

(forall ((relation ?r))
  (abridgment-relation [?r ?r]))
(forall ((relation ?r) (relation ?s))
  (=> (and (abridgment-relation [?r ?s]) (abridgment-relation [?s ?r]))
    (= ?r ?s)))
(forall ((relation ?r1) (relation ?r2) (relation ?r3))
  (=> (and (abridgment-relation [?r1 ?r2]) (abridgment-relation [?r2 ?r3]))
    (abridgment-relation [?r1 ?r3])))

```

Category Theory. Two type relations are composable when the codomain of the first is equal to the domain of the second. Two type relations form a *composable pair* when they are composable. We name the component factors of the composable endorelation.

```

(iff:set composable-pair)
(forall ((composable-pair ?rs))
  (exists ((relation ?r) (relation ?s))
    (= ?rs [?r ?s])))
(forall ((relation ?r) (relation ?s))
  (<=> (composable-pair [?r ?s])
    (= (set1 ?r) (set0 ?s))))

(iff:function factor0)
(= (iff:source factor0) composable-pair)
(= (iff:target factor0) relation)
(forall ((relation ?r) (relation ?s) (composable-pair [?r ?s]))
  (= (factor0 [?r ?s]) ?r))

(iff:function factor1)
(= (iff:source factor1) composable-pair)
(= (iff:target factor1) relation)
(forall ((relation ?r) (relation ?s) (composable-pair [?r ?s]))
  (= (factor1 [?r ?s]) ?s))

```

The *composition* of a composable pair of type relations $r : X \rightarrow Y$ and $s : Y \rightarrow Z$ is a type relation $r \circ s : X \rightarrow Z$. The domain of the composite is the domain of the first factor, and the codomain of the composite is the codomain of the second factor.

```

(iff:function composition)
(= (iff:source composition) composable-pair)
(= (iff:target composition) relation)
(forall ((relation ?r) (relation ?s) (composable-pair [?r ?s]))
  (and (= (set0 (composition [?r ?s])) (set0 ?r))
    (= (set1 (composition [?r ?s])) (set1 ?s))))

```

Composition is associative.

```
(forall ((relation ?r) (relation ?s) (relation ?t))
  (composable-pair [?r ?s]) (composable ?s ?t))
  (= (composition [?r (composition [?s ?t])])
    (composition [(composition [?r ?s]) ?t])))
```

Composition is surjective (see identity properties below).

```
(forall ((relation ?t))
  (exists ((relation ?r) (relation ?s) (composable-pair [?r ?s]))
    (= (composition [?r ?s]) ?t)))
```

The pointwise definition of the composition $r \circ s : X \rightarrow Z$ of two composable relations $r : X \rightarrow Y$ and $s : Y \rightarrow Z$ is given by $r \circ s = \{(x, z) \mid x \in X, z \in Z, \exists y \in Y r(x, y) \& s(y, z)\}$.

```
(forall ((relation ?r) (relation ?s) (composable-pair [?r ?s])
  ?x ((set0 ?r) ?x) ?z ((set1 ?s) ?z))
  (<=> ((composition [?r ?s]) ?x ?z)
    (exists ((set1 ?r) ?y)
      (and (?r ?x ?y) (?s ?y ?z)))))
```

For every type set X , there is an associated *identity* type relation $1_X : X \rightarrow X$ with X as domain and codomain.

```
(iff:function identity)
(= (iff:source identity) type.set:set)
(= (iff:target identity) relation)
(forall ((type.set:set ?X))
  (and (= (set0 (identity ?X)) ?X)
    (= (set1 (identity ?X)) ?X)))
```

The identity satisfies two unit laws with respect to composition.

```
(forall ((relation ?r))
  (and (= (composition [(identity (set0 ?r)) ?r]) ?r)
    (= ?r (composition [?r (identity (set1 ?r))])))
```

Identity is injective; hence, sets can be regarded as special relations that satisfy the unit laws.

```
(forall ((type.set:set ?X0) (type.set:set ?X1))
  (= (identity ?X0) (identity ?X1)))
(= ?X0 ?X1)
```

The pointwise definition of the identity $1_X : X \rightarrow X$ of a set X is given by $1_X = \{(x, x) \mid x \in X\}$.

```
(forall ((type.set:set ?X) (?X ?x0) (?X ?x1))
  (<=> ((identity ?X) ?x0 ?x1)
    (= ?x0 ?x1)))
```

Conversion.

$$\begin{array}{ccc}
 \text{rel} & \xrightarrow{(-)} & \text{pred} \\
 \sigma_{01} \downarrow & & \downarrow \gamma \\
 \text{set}^2 & \xrightarrow{x} & \text{set}
 \end{array}$$

Each type relation r embeds as a type *predicate* \hat{r} , hence a type injective *function*. Thus, there is a predicate map $(\hat{-}) : \text{rel} \rightarrow \text{pred}$. More specifically, each type relation $r : X_0 \rightarrow X_1$ is a type injective *function* (more generally, a monomorphism) $r : \text{ext}(r) \hookrightarrow X_0 \times X_1$. The source (target) of the injective function is the extent (product of the component set pair) of the relation r . The product of the component sets is the genus of the predicate of a relation, and the extent is the differentia.

```
(iff:function function)
(= (iff:source function) relation)
(= (iff:target function) type.ftn:function)
(forall ((relation ?r))
  (and (= (type.ftn:source (function ?r)) (extent ?r))
        (= (type.ftn:target (function ?r)) (type.lim.prd2.obj:product (set-pair ?r)))
        (type.ftn:injection (function ?r)))

(forall ((relation ?r1) (relation ?r2))
  (=> (= (function ?r1) (function ?r2))
       (= ?r1 ?r2)))

(iff:function predicate)
(= (iff:source predicate) relation)
(= (iff:target predicate) type.pred:predicate)
(forall ((relation ?r))
  (and (= (type.pred:differentia (predicate ?r)) (extent ?r))
        (= (type.pred:genus (predicate ?r)) (type.lim.prd2.obj:product (set-pair ?r)))
        (= (type.pred:function (predicate ?r)) (function ?r))))

(forall ((relation ?r)
  ((set0 ?r) ?x0) (set1 ?r) ?x1))
  (<=> ((predicate ?r) [?x0 ?x1]) (?r ?x0 ?x1)))
```

The canon of any relation r is equal to the canon of the predicate of r .

```
(forall ((relation ?r))
  (= (canon ?r) (type.pred:canon (predicate ?r))))
```

Every type relation $r : X_0 \rightarrow X_1$ has an associated *span* $\hat{r} = (\pi_0^r : X_0 \leftarrow \text{ext}(r) \rightarrow X_1 : \pi_1^r)$, whose pairing is the injection $r = (\pi_0^r, \pi_1^r) : \text{ext}(r) \rightarrow X_0 \times X_1$.

```
(iff:function span)
(= (iff:source span) relation)
(= (iff:target span) type.lim.prd2.obj:cone)
(forall ((relation ?r))
  (and (= (type.lim.prd2.obj:function0 (span ?r)) (projection0 ?r))
        (= (type.lim.prd2.obj:function1 (span ?r)) (projection1 ?r))
        (= (type.lim.prd2.obj:vertex (span ?r)) (extent ?r))
        (= (type.lim.prd2.obj:pairing (span ?r)) (function ?r))))
```

The canon of any relation r is equal to the relation of the span of r .

```
(forall ((relation ?r))
  (= (canon ?r) (type.spn:relation (span ?r))))
```

For any relation $r : X_0 \rightarrow X_1$, the unique 2-cell of its span is called the *injection* span 2-cell $\iota_r = !_{\text{spn}(r)} : \text{spn}(r) \Rightarrow 1_{X_0, X_1}$ (Figure 12).

```
(iff:function injection)
(= (iff:source injection) relation)
```

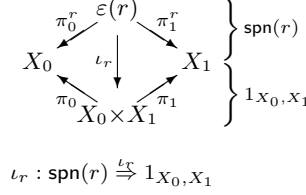


Figure 12: Injection of a Relation

```
(= (iff:target injection) type.spn.mor:2-cell)
(forall ((relation ?r))
  (and (= (type.spn.mor:source (injection ?r)) (span ?r))
        (= (type.spn.mor:target (injection ?r)) (type.spn:terminal (set-pair ?r)))
        (= (type.spn.mor:function (injection ?r)) (function ?r))
        (= (injection ?r) (type.spn:unique (span ?r)))))
```

Any type relation $r : X_0 \rightarrow X_1$ with injective function $\text{ext}(r) \xrightarrow{r} X_0 \times X_1$ has an *opposite* type relation $r^\circ : X_1 \rightarrow X_0$ with injective function $\text{ext}(r) \xrightarrow{r} X_0 \times X_1 \xrightarrow{\tau} X_1 \times X_0$, where τ is the binary product twist bijection.

```
(iff:function opposite)
(= (iff:source opposite) relation)
(= (iff:target opposite) relation)
(forall ((relation ?r))
  (and (= (set0 (opposite ?r)) (set1 ?r))
        (= (set1 (opposite ?r)) (set0 ?r))
        (= (function (opposite ?r))
            (type.ftn:composition [(function ?r) (type.lim.prd2.obj:twist (set-pair ?r))])))
```

The pointwise definition of the opposite is: $r^\circ(x_1, x_0)$ when $r(x_0, x_1)$.

```
(forall ((relation ?r) ((set0 ?r) ?x0) ((set1 ?r) ?x1))
  (<=> ((opposite ?r) ?x1 ?x0) (?r ?x0 ?x1)))
```

The opposite is an involution: $r^{\circ\circ} = r$, $(r \circ s)^\circ = s^\circ \circ r^\circ$, and $1_X^\circ = 1_X$.

```
(forall ((relation ?r))
  (= (opposite (opposite ?r)) ?r))

(forall ((relation ?r) (relation ?s) (composable-pair [?r ?s]))
  (= (opposite (composition [?r ?s]))
      (composition [(opposite ?s) (opposite ?r)])))

(forall ((type.set:set ?X))
  (= (opposite (identity ?X)) (identity ?X)))
```

For any type relation $r : X_0 \rightarrow X_1$, there are two *fiber* type functions $\varphi_r^{01} : X_0 \rightarrow \wp X_1$ and $\varphi_r^{10} : X_1 \rightarrow \wp X_0$, where the 01-fiber is defined by $\varphi_r^{01} = \varphi_{\pi_0^r} \cdot \wp \pi_1^r : X_0 \rightarrow \wp \text{ext}(r) \rightarrow \wp X_1$ and the 10-fiber is defined dually.

```
(iff:function fiber01)
(= (iff:source fiber01) relation)
```

```

(= (iff:target fiber01) type.ftn:function)
(forall ((relation ?r))
  (and (= (type.ftn:source (fiber01 ?r)) (set0 ?r))
        (= (type.ftn:target (fiber01 ?r)) (type.set:power (set1 ?r)))
        (= (fiber01 ?r)
            (type.ftn:composition
              [(type.ftn:fiber (projection0 ?r))
               (type.ftn:power (projection1 ?r))])))

(iff:function fiber10)
(= (iff:source fiber10) relation)
(= (iff:target fiber10) type.ftn:function)
(forall ((relation ?r))
  (and (= (type.ftn:source (fiber10 ?r)) (set1 ?r))
        (= (type.ftn:target (fiber10 ?r)) (type.set:power (set0 ?r)))
        (= (fiber10 ?r)
            (type.ftn:composition
              [(type.ftn:fiber (projection1 ?r))
               (type.ftn:power (projection0 ?r))])))

```

The pointwise definition of the 01-fiber is $\varphi_r^{01}(x_0) = \{x_1 \in X_1 \mid r(x_0, x_1)\}$, and the 10-fiber is defined dually.

```

(forall ((relation ?r) ((set0 ?r) ?x0) ((set1 ?r) ?x1))
  (<=> (((fiber01 ?r) ?x0) ?x1)
        (?r ?x0 ?x1))

(forall ((relation ?r) ((set1 ?r) ?x1) ((set0 ?r) ?x0))
  (<=> (((fiber10 ?r) ?x1) ?x0)
        (?r ?x0 ?x1))

```

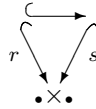
The two fibers are equivalent under involution.

```

(forall ((relation ?r))
  (and (= (fiber01 (opposite ?r)) (fiber10 ?r))
        (= (fiber10 (opposite ?r)) (fiber01 ?r))))

```

Subobject. For any two type relations $r : X_0 \rightarrow X_1$ and $s : Y_0 \rightarrow Y_1$, r is *included in* s , denoted by $r \subseteq s$, when (the span of) r belongs to (the span of) s ; equivalently, when the predicate of r is included in the predicate of s ; equivalently, when the function of r belongs to the function of s . When r is included in s , the set pair of r is the set pair of s , $(X_0, X_1) = (Y_0, Y_1)$. The inclusion endorelation on relations is a preorder (reflexive and transitive). We name the component *parts* of an inclusion relationship. There are projections $\pi_0^{\subseteq}, \pi_1^{\subseteq} : \text{ext}(\subseteq) \rightarrow \text{rel}$ to the component parts.



subobject
preorder $\wp_{\text{Set}}(\bullet \times \bullet) = \langle \wp_{\text{Set}}(\bullet \times \bullet), \subseteq \rangle$

```

(iff:set inclusion-relation)
(forall ((inclusion-relation ?rs))
  (exists ((relation ?r) (relation ?s))
    (= ?rs [?r ?s])))
(forall ((relation ?r) (relation ?s))

```

```

    (<=> (inclusion-relation [?r ?s])
         (type.ftn:belonging [(function ?r) (function ?s)])))
(forall ((relation ?r) (relation ?s))
  (<=> (inclusion-relation [?r ?s])
       (type.spn:belonging [(span ?r) (span ?s)])))
(forall ((relation ?r) (relation ?s))
  (<=> (inclusion-relation [?r ?s])
       (type.pred:inclusion-relation [(predicate ?r) (predicate ?s)])))

(forall ((relation ?r) (relation ?s))
  (=> (inclusion-relation [?r ?s])
      (= (extent ?r) (extent ?s))))

(forall ((relation ?r))
  (inclusion-relation [?r ?r]))
(forall (relation ?r) (relation ?s) (relation ?t))
  (=> (and (inclusion-relation [?r ?s]) (inclusion-relation [?s ?t]))
      (inclusion-relation [?r ?t])))

(iff:function part0)
(= (iff:source part0) inclusion-relation)
(= (iff:target part0) relation)
(forall ((relation ?r) (relation ?s) (inclusion-relation [?r ?s]))
  (= (part0 [?r ?s]) ?r))

(iff:function part1)
(= (iff:source part1) inclusion-relation)
(= (iff:target part1) relation)
(forall ((relation ?r) (relation ?s) (inclusion-relation [?r ?s]))
  (= (part1 [?r ?s]) ?s))

```

Inclusion on relations generalizes inclusion on the powerset $X_0 \times X_1$. We can recapture the latter through the canonically strict associate: for any two type relations r and s , $r \subseteq s$ iff $[r] \subseteq [s]$.

```

(forall ((relation ?r) (relation ?s))
  (<=> (inclusion-relation [?r ?s])
       (inclusion-relation [(canon ?r) (canon ?s)])))

```

For any two type relations r and s , r is *equivalent to (isomorphic to)* s , denoted by $r \equiv s$, when r and s are included in each other, $r \subseteq s$ and $s \subseteq r$. When r is equivalent to s , the extent of r is isomorphic to the extent of s , $\varepsilon(r) \cong \varepsilon(s)$, and r and s factor through each other via the inverses. For any two type relations r and s , r is equivalent to s iff the canonically strict associates are equal: $r \equiv s$ iff $[r] = [s]$. The equivalence endorelation on type relations is an equivalence relation (reflexive, symmetric and transitive). Note: for any type relation r , the collection of type relations equivalent to r , the equivalence class $[r]$, is not necessarily locally small.

```

(iff:set equivalence) (iff:set isomorphism) (= isomorphism equivalence)
(forall ((equivalence ?rs))
  (exists ((relation ?r) (relation ?s))
    (= ?rq [?r ?s])))
(forall ((relation ?r) (relation ?s))
  (<=> (equivalence [?r ?s])
       (and (inclusion-relation [?r ?s])
            (inclusion-relation [?s ?r]))))
(forall ((relation ?r) (relation ?s))

```

```

(<=> (equivalence [?r ?s])
      (= (canon ?r) (canon ?s))))

(forall ((relation ?r))
  (equivalence [?r ?r]))
(forall ((relation ?r) (relation ?s))
  (=> (equivalence [?r ?s]) (equivalence [?s ?r])))
(forall ((relation ?r) (relation ?s) (relation ?r))
  (=> (and (equivalence [?r ?s]) (equivalence [?s ?r]))
      (equivalence [?r ?r])))

```

Any relation is equivalent to its canonically strict associate, $r \equiv [r]$. Hence, for any type relation r , the equivalence class $[r]$ has the canon as canonical representative.

```

(forall ((relation ?r))
  (equivalence [?r (canon ?r)]))

```

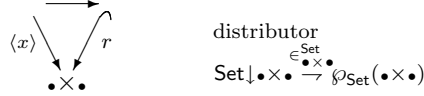
Inclusions compose: for composable pairs of inclusions $r \subseteq r' : X \rightarrow Y$ and $s \subseteq s' : Y \rightarrow Z$, composition is an inclusion $r \circ s \subseteq r' \circ s' : X \rightarrow Z$.

```

(forall ((relation ?r) (relation ?rp) (relation ?s) (relation ?sp)
  (composable-pair [?r ?s]) (composable ?rp ?sp))
  (=> (and (inclusion-relation [?r ?rp]) (inclusion-relation [?s ?sp]))
      (inclusion-relation [(composition [?r ?s]) (composition [?rp ?sp])))))

```

For any type span $x = (\bullet \xleftarrow{x_0} \xrightarrow{x_1} \bullet)$ and any type relation $\xrightarrow{r} \bullet \times \bullet$, x is a *member* of r , $x \in r$, when x belongs to (the span of) r ; that is, when there is a proof function h such that $\langle x \rangle = h \cdot r$. As noted when defining the belonging relation, the proof function h is unique. The membership relation from type spans to type relations is closed with respect to span belonging on the left and relation inclusion on the right. We name the component *span* and (relational) *part* of a membership relationship. There are projections $\pi_0^\in \text{ : ext}(\in) \rightarrow \text{spn}$ and $\pi_1^\in \text{ : ext}(\in) \rightarrow \text{rel}$ to the components.



```

(iff:set membership)
(forall ((membership ?xr))
  (exists ((type.spn:span ?x) (relation ?r))
    (= ?xr [?x ?r])))
(forall ((type.spn:span ?x) (relation ?r))
  (<=> (membership [?x ?r])
      (type.spn:belonging [?x (span ?r)])))

(forall ((type.spn:span ?x) (relation ?r) (membership [?x ?r]))
  (and (forall ((type.spn:span ?y) (type.spn:belonging [?y ?x]))
    (membership [?y ?r]))
    (forall ((relation ?s) (inclusion [?r ?s]))
    (membership [?x ?s])))))

(iff:function element)
(= (iff:source element) membership)
(= (iff:target element) type.spn:span)

```

```

(forall ((type.spn:span ?x) (relation ?r) (membership [?x ?r]))
  (= (element [?x ?r]) ?x))

(iff:function part)
(= (iff:source part) membership)
(= (iff:target part) relation)
(forall ((type.spn:span ?x) (relation ?r) (membership [?x ?r]))
  (= (part [?x ?r]) ?r))

```

Fact 3 *Inclusion is equivalent to universal implication of membership*

$$r \subseteq s \text{ iff } \forall_{x \in (X_0, X_1)} (x \in r \Rightarrow x \in s) \quad \subseteq = \in \setminus \in$$

Hence, the usual relationship holds between inclusion and membership.

```

(forall ((relation ?r) (relation ?s))
  (<=> (inclusion [?r ?s])
    (forall (?x (type.spn:span ?x))
      (=> (membership [?x ?r]) (membership [?x ?s])))))

```

We name the unique *proof* function for membership.

```

(iff:function proof)
(= (iff:source proof) membership)
(= (iff:target proof) type.ftn:function)
(forall ((type.spn:span ?x) (relation ?r) (membership [?x ?r]))
  (and (= (type.ftn:source (proof [?x ?r])) (type.spn:vertex ?x))
    (= (type.ftn:target (proof [?x ?r])) (extent ?r))
    (= (function ?x) (type.ftn:composition [(proof [?x ?r]) (function ?r)]))))

```

For any two relations $r : X_0 \rightarrow X_1$ and $s : X_0 \rightarrow X_1$ that share common domain and codomain sets, there is a meet relation $r \wedge s : X_0 \rightarrow X_1$ whose extent and projection functions are defined in terms of the pullback of the pairing functions. The pullback projections are the functions of 2-cells $\pi_0 : \text{spn}(r \wedge s) \xrightarrow{\pi_0} \text{spn}(r)$ and $\pi_1 : \text{spn}(r \wedge s) \xrightarrow{\pi_1} \text{spn}(s)$, showing that the meet is included in each component: $r \wedge s \subseteq r$ and $r \wedge s \subseteq s$. Any other relation included in both r and s also is included in the meet $r \wedge s$.

```

(forall ((relation ?r) (relation ?s) (= (set-pair ?r) (set-pair ?s)))
  (and (= (extent (meet [?r ?s])) (type.lim.pbk.obj:pullback [(function ?r) (function ?s)]))
    (= (projection0 (meet [?r ?s]))
      (type.ftn:composition
        [(type.lim.pbk.obj:projection0 [(function ?r) (function ?s)] (projection0 ?r)]))
      (= (projection1 (meet [?r ?s]))
        (type.ftn:composition
          [(type.lim.pbk.obj:projection1 [(function ?r) (function ?s)] (projection1 ?s)]))))))

(forall ((relation ?r) (relation ?s) (= (set-pair ?r) (set-pair ?s)))
  (and (inclusion-relation [(meet [?r ?s]) ?r])
    (inclusion-relation [(meet [?r ?s]) ?s])
    (forall ((relation ?w) (= (set-pair ?w) (set-pair ?r)))
      (=> (and (inclusion-relation [?w ?r]) (inclusion-relation [?w ?s]))
        (inclusion-relation [?w (meet [?r ?s])]))))

```

1.6.1 Type Endorelations

Basics. The set of all type *endorelations* is an IFF set. Type endorelations are special type set relations, whose domain and codomain sets are identical.

```
(iff:set endorelation)
(forall ((endorelation ?r)) (relation ?r))
(forall ((relation ?r))
  (<=> (endorelation ?r)
      (= (set0 ?r) (set1 ?r))))
```

There is a common component type *set*.

```
(iff:function set)
(= (iff:source set) endorelation)
(= (iff:target set) type.set:set)
(forall (?r (endorelation ?r))
  (and (= (set ?r) (set0 ?r))
        (= (set ?r) (set1 ?r))))
```

Order Theory. We have predicates to express the fact that a type endorelation is *reflexive*, *transitive*, *antisymmetric* or *symmetric*. These are predicates (adjectives) that refer to (modify) endorelations.

- A emphreflexive type endorelation is one that contains the identity endorelation $1_X \subseteq r$. Since there is no notion of predicate specified at the IFF level, at the type level we use the differentia set of the reflexive relation¹³ to indirectly specify it.
- A emphtransitive type endorelation is one that contains the relational composition of it with itself $r \circ r \subseteq r$.
- A emphsymmetric type endorelation is one whose transpose is contained in it $r^\times \subseteq r$.
- An *antisymmetric* type endorelation is one where the meet of it with its transpose is a subrelation of the identity $r \wedge_X r^\times \subseteq 1_X$.

```
(iff:set reflexive-relation)
(forall ((reflexive-relation ?r)) (endorelation ?r))
(forall ((endorelation ?r))
  (<=> (reflexive-relation ?r)
      (inclusion-relation [(identity (set ?r)) ?r])))
```

```
(iff:set transitive-relation)
(forall ((transitive-relation ?r)) (endorelation ?r))
(forall ((endorelation ?r))
  (<=> (transitive-relation ?r)
      (inclusion-relation [(composition [?r ?r]) ?r])))
```

```
(iff:set symmetric-relation)
(forall ((symmetric-relation ?r)) (endorelation ?r))
```

¹³We use this idea of “differentia serving as proxy” for other predicates. Just as for binary relations, this is a small part of the bootstrapping mechanism of the IFF metashell.

```

(forall ((endorelation ?r))
  (<=> (symmetric-relation ?r)
    (forall (((set ?r) ?x0) ((set ?r) ?x1))
      (inclusion-relation [(opposite ?r) ?r])))

(iff:set antisymmetric-relation)
(forall ((antisymmetric-relation ?r) (endorelation ?r))
  (forall ((endorelation ?r))
    (<=> (antisymmetric-relation ?r)
      (inclusion-relation [(meet [?r (opposite ?r)]) (identity (set ?r))])))

```

We can declare special type endorelations called *preorders*, *partial-orders* and *equivalence relations*. Preorders are reflexive and transitive. Partial orders are preorders that are antisymmetric. Equivalence relations are reflexive, symmetric and transitive.

```

(iff:set preorder)
(forall ((preorder ?r) (endorelation ?r))
  (forall ((endorelation ?r))
    (<=> (preorder ?r)
      (and (reflexive-relation ?r) (transitive-relation ?r))))

(iff:set partial-order)
(forall ((partial-order ?r) (preorder ?r))
  (forall ((preorder ?r))
    (<=> (partial-order ?r)
      (antisymmetric-relation ?r)))

(iff:set equivalence-relation)
(forall ((equivalence-relation ?r) (preorder ?r))
  (forall ((preorder ?r))
    (<=> (equivalence-relation ?r)
      (symmetric-relation ?r)))

```

Let $\equiv : X \rightarrow X$ be an equivalence relation on X with two projection functions $\pi_0^\equiv, \pi_1^\equiv : \text{ext}(\equiv) \rightarrow X$. There is a quotient set $\text{quo}(\equiv) = X/\equiv = \{[x]_\equiv \mid x \in X\}$ and a (*canon*)ical surjection $[-]_\equiv : X \rightarrow \text{quo}(\equiv)$, where $[x]_\equiv = \{x' \in X \mid x' \equiv x\}$ is the equivalence class of $x \in X$. The quotient is defined to be the range of the 01-fiber of the relation \equiv and the canon is define to be the surjective-factor of the 01-fiber

$$\phi_{01}^\equiv : X \xrightarrow{[-]_\equiv} \text{quo}(\equiv) \rightarrow \wp X.$$

```

(iff:function quotient)
(= (iff:source quotient) equivalence-relation)
(= (iff:target quotient) type.set:set)
(forall (?e (equivalence-relation ?e))
  (= (quotient ?e) (type.ftn:range (fiber01 ?e))))

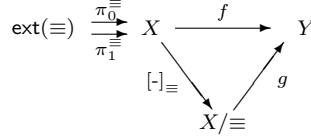
(iff:function canon)
(= (iff:source canon) equivalence-relation)
(= (iff:target canon) type.ftn:function)
(forall (?e (equivalence-relation ?e))
  (and (= (type.ftn:source (canon ?e)) (set ?e))
    (= (type.ftn:target (canon ?e)) (quotient ?e))
    (= (canon ?e) (type.ftn:surjective-factor (fiber01 ?e)))))

```

A function $f : X \rightarrow Y$ *respects* an equivalence relation $\equiv : X \rightarrow X$ when (1) the source of f is the set of \equiv , $\partial_0(f) = \sigma(\equiv)$, and (2) composition with the projections gives the same function, $\pi_0^\equiv \cdot f = \pi_1^\equiv \cdot f$ (that is, $x_1 \equiv x_2$ implies $f(x_1) = f(x_2)$ for any pair $x_1, x_2 \in X$).

```
(iff:set respects-relation)
(forall ((respects-relation ?fe))
  (exists ((type.ftn:function ?f) (equivalence-relation ?e))
    (= ?fe [?f ?e])))
(forall ((type.ftn:function ?f) (equivalence-relation ?e))
  (<=> (respects-relation [?f ?e])
    (and (= (source ?f) (set ?e))
      (type.ftn:composition [(projection0 ?e) ?f])
      (type.ftn:composition [(projection1 ?e) ?f])))))
```

If a function $f : X \rightarrow Y$ respects an equivalence relation $\equiv : X \rightarrow X$, then f factors through the quotient: there is a function $g : X/\equiv \rightarrow Y$ such that $f = [-]_\equiv \cdot g$.



```
(forall ((type.ftn:function ?f) (equivalence-relation ?e) (respects-relation [?f ?e]))
  (exists ((type.ftn:function ?g)
    (and (= (type.ftn:source ?g) (quotient ?e))
      (= (type.ftn:target ?g) (type.ftn:target ?f))
      (= ?f (type.ftn:composition [(canon ?e) ?g])))))
```

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.