

# The IFF Type Namespace

Robert E. Kent

December 20, 2007

## Contents

<b>1</b>	<b>The Core Component</b>	<b>2</b>
1.1	Introduction . . . . .	2
0.1	Type Sets . . . . .	2
0.2	Type Functions . . . . .	14
0.2.1	Function Morphisms . . . . .	28
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

## 0.2 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 morphisms<sup>4</sup>. 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,  $\sigma(x) = \partial_1(x)$

<sup>4</sup>Note 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.

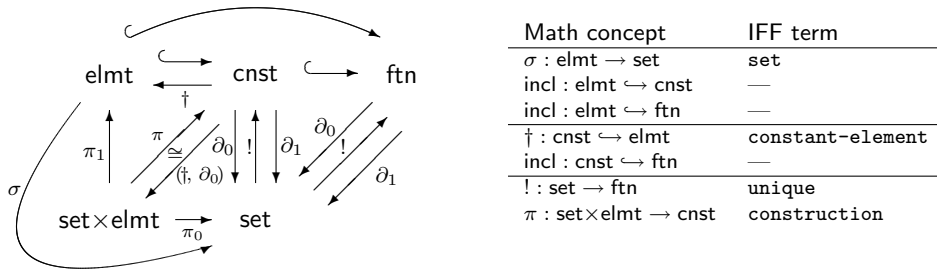


Figure 1: Basic Function Kinds

(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  $\text{cnst}$  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))
```

A type function  $f : X \rightarrow Y$  is relatively *small* when its source and target sets are both relatively small (meta) sets. That is, the predicate “small” has as its genus the set of all type functions and as its differentia the set of all meta functions.

```
(iff:set small-function)
(forall ((small-function ?f)) (function ?f))
(forall ((function ?f))
  (<=> (small-function ?f)
    (and (type.set:small-set (source ?f))
      (type.set:small-set (target ?f)))))
(forall ((function ?f))
  (<=> (small-function ?f)
    (meta.ftn:function ?f)))
```

**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))
  (exists ((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 set  $\xrightarrow{1}$  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 *appliant* when the set of the element is equal to the source of the function,  $\sigma(x) = \partial_0(f)$ . The IFF set of all appliant pairs is a(n implicit) subset of the IFF set of all composable pairs. We name the projection *factors* of appliant pairs.

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

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

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

The type set of all appliant 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) (appliant-pair [?x ?f]))
  (composable-pair [?x ?f]))
```

The *application* of an appliant 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 appliant pair  $x \in X$  and  $f : X \rightarrow Y$ ; that is, application is a restriction of composition.

```
(iff:function application)
(= (iff:source application) appliant-pair)
(= (iff:target application) element)
(forall ((element ?x) (function ?f) (appliant-pair [?x ?f]))
  (= (set (application [?x ?f])) (target ?f)))
(forall ((element ?x) (function ?f) (appliant-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  $[x]_{\equiv_f} = \{x' \in X \mid f(x') = f(x)\}$  is the equivalence class of  $x$  with respect to the kernel of  $f$ , and the *coequalizer*<sup>5</sup> 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))))
```

<sup>5</sup>So named because it is the coequalizer of the kernel projection pair.

```

(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  $f = [-]_f \cdot \wr_f$  for some (injective) function  $\text{coapply}_f = \wr_f : \text{coim}_f \rightarrow Y$ . This factor, called the *coapplication* of  $f$ , is defined by  $\wr_f([x]_{\equiv f}) = f(x)$ .

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

```

(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  $d \cdot m = \wr_f$ .

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

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))))

```

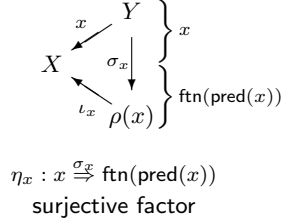


Figure 2: Unit Function 2-Cell

For any type function  $x : Y \xrightarrow{x} X$ , there is a *unit* function 2-cell  $\eta_x : x \Rightarrow \text{ftm}(\text{pred}(x))$  whose source is  $x$ , whose target  $\text{ftm}(\text{pred}(x))$  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))))
```

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 an 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])
```

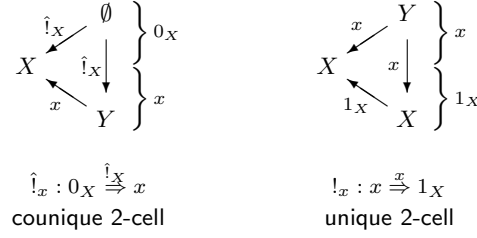


Figure 3: Counique and Unique Function 2-Cells

```

(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

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

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  $0_X = \emptyset \xrightarrow{i_X} X$ , and the identity function forms a *terminal* (predicative) function  $1_X = X \xrightarrow{1_X} X$ .

```

(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  $\hat{!}_x : 0_X \Rightarrow x$  and a *unique* type function 2-cell  $!_x : x \Rightarrow 1_X$  (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))))))

```

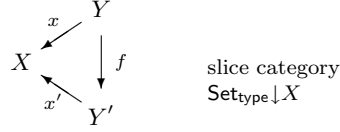


Figure 4: Function 2-Cell

### 0.2.1 Function Morphisms

**Basics.** A type function *morphism* is a morphism in the comma category<sup>6</sup>

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

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)  $x, x' \in \mathbf{fn}_{\text{type}}$ ,  $x$  *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* function<sup>7</sup>  $p \in \mathbf{fn}_{\text{type}}$  such that  $x = p \cdot x'$ . We name the component *elements*

<sup>6</sup>The comma category  $(1 \downarrow 1) = \mathbf{Set}_{\text{type}}^{\rightarrow}$  is called the arrow category of  $\mathbf{Set}_{\text{type}}$ .

<sup>7</sup> $p$  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))])))

```

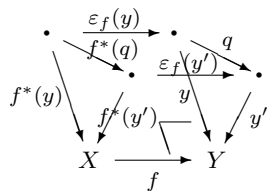
The identity map is injective  $\text{ftn} \xrightarrow{1} \text{mor}$ . Hence, functions can be regarded as special 2-cells that satisfy the unit laws.

```

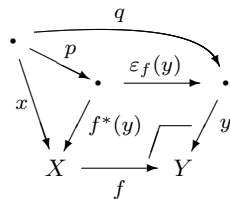
(forall ((type.ftnt:function ?x0) (type.ftnt:function ?x1)
  (= (identity ?x0) (identity ?x1)))
  (= ?x0 ?x1))

```

**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  $\text{Set}_{\text{type}}$  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  $y^{\text{th}}$  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  $\varepsilon_f$



adjunction  $\Sigma_f \dashv f^*$

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

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

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.