The IFF Basic KIF Ontology

The IFF Basic KIF Ontology is situated at the top metalevel – the highest level of the IFF Foundation Ontology (the SUO metalevel). Its purpose is to provide an interface between the KIF logical language and the SUO ontological structure. Principally, it does this by servicing the upper metalevel. It contains rudimentary (fundamental) namespaces for collections, functions, relations, limits and colimits. The IFF Basic KIF Ontology provides an adequate foundation for representing ontologies in general and for defining other metalevel ontologies in particular. The following description of the IFF Basic KIF Ontology follows Mac Lane's beginning axiomatization for category theory (Mac Lane, 1971) in that it introduces terminology and provides an axiomatization for this terminology, but it does not give a formal interpretation using set theory – it only gives an informal, intuitive interpretation.

Table 1 lists all 72 terms (63 concepts or non-identical terms) in the Basic KIF Ontology, partitioned according to whether the term is a collection, relation or function. Terms in boldface are used in the IFF Core Ontology with basic terms underlined. Although the IFF Basic KIF Ontology is the highest and most generic module, it is also the least detailed. Terminology has been placed in the IFF Basic KIF Ontology only when it is needed in the IFF upper metalevel[§]. All upper metalevel ontologies (Core, Classification and Category Theory) import and use, either directly or indirectly, the IFF Basic KIF Ontology.

Table 1: Terms introduced in the Basic KIF Ontology

	Collection	Relation	Function	Example
KIF	**collection** pair triple pair-collection triple-collection tuple-collection **sub-collection**	**subcollection** **disjoint** **isomorphic**	**binary-union** **binary-intersection** element arity type	**zero** = **nothing** = **null** = **empty** **one** = **unit** **two** **three**
	relation total functional total-functional	subrelation **abridgment**	**collection1** **collection2** **extent**	
	partial-function **function** = tuple pair-function tuple-function **injection** **surjection** bijection	**restriction** partial-restriction	**source** **target** **domain** domain-restriction identity tuple-source tuple-target **fiber** **constant** **inclusion** pfn2rel **fn2rel**	
	span **opspan**		**unique** **counique** **binary-product** binary-coproduct = binary-sum pairing copairing ternary-product **product** **coproduct** = **sum** = **disjoint-union** tupling cotupling **power** = **exponent** **pullback** pullback-pairing	

[§] The 'abridgment' term was placed here, in order to be able to precisely express the relationship between relations in the upper metalevel to their counterparts in this ontology ('subclass', 'disjoint' and 'restriction'). The 'pullback' term was placed here, since this is needed in the Core Ontology to define the two conversions of functions to relations based upon a preorder (which in turn is used to define the instance and type embedding relations in the Classification Ontology). The 'partial-function' was placed here, in order to be able to express (in a simple fashion) in the Core Ontology the 'pairing' operator for pullbacks (which in turn is used in many different places in the Category Theory Ontology; for example, with the composition opspan).

In general, type signatures are needed when valence is variable. In this ontology, signatures are not needed for either KIF functions or KIF relations, since all functions are unary and all relations are binary. In place of signatures are the source/target collections of functions and the two component collections of relations. The advantage for not requiring a signature is elimination of the dependency on sequences and natural numbers. They are simply not needed here. In the IFF Model Theory Ontology, there will appear a signature concept that corresponds to the signatures of Chris Menzel's Basic Ontology. KIF functions are (conceptually) unary, binary or ternary. The few KIF functions that are not conceptually unary are actually unary with a source KIF collection that is a binary product, a ternary product or a pullback.

Table 2 lists the numbers of times terms are used in the three ontologies in the upper metalevel- the Core Ontology, Category Theory Ontology and Classification Ontology. These are partitioned as basic, special and other terms. Basic terms are those one would expect to see in a root-level category-theoretic ontology. The special terms are used to specify tupling and cotupling in the Core Ontology.

Table 2: Frequency of Term Use in the Upper Metalevel

	Core		Category Theory		Classification	
	Term	Freq	Term	Freq	Term	Freq
Basic	function	392	function	234	function	255
	source	386	source	221	source	241
	target	386	target	221	target	241
	collection	116	collection	29	collection	49
	restriction	45	restriction	4	restriction	0
	subcollection	38	subcollection	17	subcollection	4
	pullback	27	pullback	6	pullback	17
	relation	27	relation	5	relation	11
	collection1	25	collection1	4	collection1	11
	collection2	25	collection2	4	collection2	11
	extent	20	extent	4	extent	11
	opspan	10	opspan	3	opspan	8
	abridgment	9	abridgment	0	abridgment	0
Special	partial-function	18	partial-function	0	partial-function	0
	domain	28	domain	0	domain	0
	power	25	power	1	power	0
Other	three	14				
	binary-product	9				
	two	7				
	binary-intersection	5				
	isomorphic	3				
	fiber	3				
	identity	3				
	product coproduct disjoint sub-collection binary-union fn2rel unique counique constant inclusion injection surjection empty unit	1				

Table 3 lists the correspondence between standard mathematical notation and the ontological terminology in the namespace for logic. Table 4 lists the correspondence between standard mathematical notation and the ontological terminology in the IFF Basic KIF Ontology.

Table 3: Correspondences – math and logic

Math	Ontological Terminology	Natural Language Description
$\forall$	`forall`	universal quantifier
$\exists$	`exists`	existential quantifier
$\wedge$	`and`	conjunction
$\vee$	`or`	disjunction
$\neg$	`not`	negation
$\rightarrow$	`=>`	implication
$\leftrightarrow$	`<=>`	equivalence

Table 4: Correspondences - math and ontology

Math	Ontological Terminology	Natural Language Description
C	`collection`	collection

$\boldsymbol{f} : C_1 \rightarrow C_2$	`function`
	`source`
	`target`
		function
		source (domain)
		target (codomain)

$\boldsymbol{R}$
$R \subseteq C_1 \times C_2$
	`relation`
	`extent`
	`collection1`
	`collection2`
		relation
		extent
		component collections

$(a_1, \ldots, a_n)$	`[?a1 … ?an]`	sequence notation

Collections

- A collection is a generic set, whether a small set, a class or something bigger. The main KIF collections are *collection*, *relation*, *partial function* and *function*. The relations are binary. The partial functions have a single source, which could be a binary product, a ternary product, or a pullback.

```
(1) (collection collection)

(2) (collection relation)

(3) (collection partial-function)

(4) (collection function)
```

- Collections, relations and partial functions are pair-wise disjoint. Functions are special kinds of partial functions.

```
(5) (disjoint collection relation)

(6) (disjoint collection partial-function)

(7) (disjoint relation partial-function)

(8) (subcollection function partial-function)
```

- Only collections or relations can be predicated of other things.

```
 (9) (forall (?p @args)
         (=> (?p @args)
             (or (collection ?p) (relation ?p))))
```

- Only functions can be applied to other things.

```
(10) (forall (?f ?x @args)
         (=> (= ?x (?f @args))
             (function ?f)))
```

- A collection is essentially a unary predicate, in that it is never true of multiple things; only objects, rather than *n*-tuples of objects for $n > 2$, are in its extension.

```
(11) (forall (?c (collection ?c))
         (not (exists (?x ?y @args)
                  (?c ?x ?y @args))))
```

- A relation is essentially a binary predicate, in that it is only true of pairs of things; that is, only pairs of objects, rather than *n*-tuples of objects for $n = 1$ or $n > 2$, are in its extension.

```
(12) (forall (?r (relation ?r))
         (and (not (exists (?x) (?r ?x)))
              (not (exists (?x ?y ?z @args) (?rel ?x ?y ?z @args)))))
```

- Here are some basic collections: *zero* = {} = ∅, *one* = {1}, *two* = {1, 2}, *three* = {1, 2, 3}. *Zero* and *one* have several synonyms. *Two* and *three* are often used for indexing. No thing is an instance of *zero*. Three canonical objects have been used in specifying these base collections: 1, 2 and 3.

```
(13) (collection zero)
     (collection nothing)
     (collection null)
     (collection empty)
     (= zero nothing)
     (= nothing null)
     (= null empty)
     (forall (?x) (not (zero ?x)))

(14) (collection one)
     (collection unit)
     (= one unit)
     (one 1)
     (forall (?x (one ?x)) (= ?x 1))

(15) (collection two)
```

```
     (two 1)
     (two 2)
     (forall (?x (two ?x)) (or (= ?x 1) (= ?x 2)))

(16) (collection three)
     (three 1)
     (three 2)
     (three 3)
     (forall (?x (three ?x)) (or (= ?x 1) (= ?x 2) (= ?x 3)))
```

- A collection C_1 is a *subcollection* of a collection C_2 when every instance of C_1 is an instance of C_2.

```
(17) (relation subcollection)
     (= (collection1 subcollection) collection)
     (= (collection2 subcollection) collection)
     (forall (?c1 (class ?c1) ?c2 (class ?c2))
         (<=> (subcollection ?c1 ?c2)
              (forall (?x) (=> (?c1 ?x) (?c2 ?x)))))
```

- Clearly, we have the following inclusions (subcollection relationships): $zero \subseteq C$ for any collection C, and $one \subseteq two \subseteq three$.

```
(18) (forall (?c (collection ?c))
         (subcollection zero ?c))
     (subcollection one two)
     (subcollection two three)
```

- The extent of the subcollection relation is named.

```
(19) (collection sub-collection)
     (subcollection sub-collection pair-collection)
     (= sub-collection (extent subcollection))
```

- One collection C_1 is *disjoint* from another collection C_2 when there is no instance of both C_1 and C_2.

```
(20) (relation disjoint)
     (= (collection1 disjoint) collection)
     (= (collection2 disjoint) collection)
     (forall (?c1 (class ?c1) ?c2 (class ?c2))
         (<=> (disjoint ?c1 ?c2)
              (forall (?x) (not (and (?c1 ?x) (?c2 ?x))))))
```

- Two collections C_1 and C_2 are *isomorphic* when there is a bijection between them.

```
(21) (relation isomorphic)
     (= (collection1 isomorphic) collection)
     (= (collection2 isomorphic) collection)
     (forall (?c1 (class ?c1) ?c2 (class ?c2))
         (<=> (isomorphic ?c1 ?c2)
              (exists (?f (bijection ?f))
                  (and (= (source ?f) ?c1)
                       (= (target ?f) ?c2)))))
```

- A *tuple* is another name for a function.

```
(22) (collection tuple)
     (= tuple function)
```

- The *arity* of a tuple is another name for its source collection (the source of the tuple as function).

```
(23) (function arity)
     (= (source arity) tuple)
     (= (target arity) collection)
     (= arity source)
```

- The *type* of a tuple is another name for its target collection (the target of the tuple as function).

```
(24) (function type)
     (= (source type) tuple)
     (= (target type) collection)
     (= type target)
```

- The *pair* collection is defined to be the (implicit) Cartesian product of everything with itself. More explicitly, a pair is a tuple with arity *two*.

```
(25) (collection pair)
     (forall (?x)
         (<=> (pair ?x)
              (and (tuple ?x)
                   (= (arity ?x) two))))
```

- We use the pairing notation ‘`[x1 x2]`’ to denote a pair of objects ‘`x1`’ and ‘`x2`’.

```
(26) (forall (?x)
         (<=> (pair ?x)
              (exists (?x1 ?x2)
                  (and (= (?x 1) ?x1)
                       (= (?x 2) ?x2)
                       (= ?x [?x1 ?x2])))))
```

- The *triple* collection is defined to be the (implicit) third Cartesian power of everything. More explicitly, a triple is a tuple with arity *three*.

```
(27) (collection triple)
     (forall (?x)
         (<=> (triple ?x)
              (and (tuple ?x)
                   (= (arity ?x) three))))
```

- We use the tripling notation ‘`[x1 x2 x2]`’ to denote a triple of objects ‘`x1`’, ‘`x2`’ and ‘`x3`’.

```
(28) (forall (?x)
         (<=> (triple ?x)
              (exists (?x1 ?x2 ?x3)
                  (and (= (?x 1) ?x1)
                       (= (?x 2) ?x2)
                       (= (?x 3) ?x3)
                       (= ?x [?x1 ?x2 ?x3])))))
```

- Pairs and triples are tuples.

```
(29) (subcollection pair tuple)

(30) (subcollection triple tuple)
```

- A *tuple of collections* is a tuple of type *collection*.

```
(31) (collection tuple-collection)
     (subcollection tuple-collection tuple)
     (forall (?c (tuple ?c))
         (<=> (tuple-collection ?c)
              (= (type ?c) collection)))
```

- The collection $collection^2 = collection \times collection$ of all pairs of collections is defined.

```
(32) (collection pair-collection}
     (forall (?c)
         (<=> (pair-collection ?c)
              (and (pair ?c)
                   (collection (?c 1))
                   (collection (?c 2)))))
```

- One can prove that *pair-collection* is the binary power of *collection*.

```
(33) (= pair-collection (power [two collection]))
```

- The collection $collection^3 = collection \times collection \times collection$ of all triples of collections is defined.

```
(34) (collection triple-collection}
     (forall (?c)
         (<=> (triple-collection ?c)
              (and (triple ?c)
                   (collection (?c 1))
                   (collection (?c 2))
                   (collection (?c 3)))))
```

- One can prove that *triple-collection* is the ternary power of *collection*.

```
(35) (= triple-collection (power [three collection]))
```

- Pair collections and triple collections are tuple collections.

```
(36) (subcollection pair-collection tuple-collection)

(37) (subcollection triple-collection tuple-collection)
```

- For any pair of collections there is a *binary union* collection and a *binary intersection* collection.

```
(38) (function binary-union)
     (= (source binary-union) pair-collection)
     (= (target binary-union) collection)
     (forall (?c1 (collection ?c1) ?c2 (collection ?c2) ?x)
         (<=> ((binary-union [?c1 ?c2]) ?x)
              (or (?c1 ?x) (?c2 ?x))))

(39) (function binary-intersection)
     (= (source binary-intersection) pair-collection)
     (= (target binary-intersection) collection)
     (forall (?c1 (collection ?c1) ?c2 (collection ?c2) ?x)
         (<=> ((binary-intersection [?c1 ?c2]) ?x)
              (and (?c1 ?x) (?c2 ?x))))
```

- For any collection C a *global element* (specializing a notion of category theory) is a function $x : 1 \rightarrow C$.

```
(40) (function element)
     (= (source element) collection)
     (= (target element) collection)
     (forall (?c (collection ?c) ?x)
         (<=> ((element ?c) ?x)
              (and (function ?x) (= (source ?x) one) (= (target ?x) ?c))))
```

- Global elements and “ordinary” elements are isomorphic. Hence, the membership notion is subsumed into the function notion.

```
(41) (forall (?c (collection ?c))
         (isomorphic ?c (element ?c)))
```

Relations

- A KIF relation $\boldsymbol{R} = \langle \mathit{collection}_1(\boldsymbol{R}), \mathit{collection}_2(\boldsymbol{R}), \mathit{extent}(\boldsymbol{R}) \rangle$ consists of three collections:
  - $\mathit{collection}_1(\boldsymbol{R})$, the first component collection,
  - $\mathit{collection}_2(\boldsymbol{R})$, the second component collection, and
  - $\mathit{extent}(\boldsymbol{R}) \subseteq \mathit{collection}_1(\boldsymbol{R}) \times \mathit{collection}_2(\boldsymbol{R})$, the extent collection.

  The extent is the collection of all pairs from the first and second component collections that satisfy the relationship. A relation is determined by the triple of its first, second and extent collections.

$$\begin{array}{c} \mathit{collection}_2(\boldsymbol{R}) \\ \uparrow \boldsymbol{R} \\ \mathit{collection}_1(\boldsymbol{R}) \end{array}$$

Figure 1: Relation

```
(42) (function collection1)
     (= (source collection1) relation)
     (= (target collection1) collection)

(43) (function collection2)
     (= (source collection2) relation)
     (= (target collection2) collection)

(44) (function extent)
     (= (source extent) relation)
     (= (target extent) collection)
     (forall (?r (relation ?r))
         (subcollection
             (extent ?r)
             (binary-product [(collection1 ?r) (collection2 ?r)])))

(45) (forall (?r (relation ?r) ?x1 ?x2)
         (<=> ((extent ?r) [?x1 ?x2])
              (and ((collection1 ?r) ?x1)
                   ((collection2 ?r) ?x2)
                   (?r ?x1 ?x2))))

(46) (forall (?r (relation ?r) ?s (relation ?s))
         (=> (and (= (collection1 ?r) (collection1 ?s))
                  (= (collection2 ?r) (collection2 ?s))
                  (= (extent ?r) (extent ?s)))
             (= r s)))
```

- One relation *r* is a *subrelation* of another relation *s* when the first and second component of *r* and *s* are the same, and the extent collection of *r* is a subcollection of the extent collection of *s*.

```
(47) (relation subrelation)
     (= (collection1 subrelation) relation)
     (= (collection2 subrelation) relation)
     (forall (?r ?s (relation ?r) (relation ?s))
         (<=> (subrelation ?r ?s)
              (and (= (collection1 ?r) (collection1 ?s))
                   (= (collection2 ?r) (collection2 ?s))
                   (subcollection (extent ?r) (extent ?s)))))
```

- One relation *r* is an *abridgment* of another relation *s* when the first component, and the second component collections of *r* are subcollections of the first component and the second component collections of *s*, respectively, and the extent of *r* is the "restriction" of the extent of *s* to the component collections of *r*. The abridgment relation is much more useful than the subrelation relation.

```
(48) (relation abridgment)
     (= (collection1 abridgment) relation)
     (= (collection2 abridgment) relation)
     (forall (?r (relation ?r) ?s (relation ?s))
         (<=> (abridgment ?r ?s)
              (and (subcollection (collection1 ?r) (collection1 ?s))
                   (subcollection (collection2 ?r) (collection2 ?s))
                   (= (extent ?r)
                      (binary-intersection
                          (extent ?s)
                          (binary-product [(collection1 ?r) (collection2 ?r)]))))))
```

- If relation r is an abridgment of relation s, then the extent of r is a subcollection of the extent of s.

```
(49) (forall (?r (relation ?r) ?s (relation ?s))
         (=> (abridgment ?r ?s)
             (subcollection (extent ?r) (extent ?s))))
```

- A relation is *total* when it satisfies the condition: every object in the first component collection is the first component of some pair in the extent of the relation. A relation is *functional* when it satisfies the condition: if the relation holds for pairs p_1 and p_2 with the same first elements, then the second elements must be identical as well (that is, p_1 and p_2 must be the same pairs). A relation is *total functional* when it is both total and functional.

```
(50) (collection total)
     (subcollection total relation)
     (forall (?r (relation ?r))
         (<=> (total ?r)
              (forall (?x1 ((collection1 ?r) ?x1))
                  (exists ?x2 ((collection2 ?r) ?x2))
                      (?r ?x1 ?x2)))))

(51) (collection functional)
     (subcollection functional relation)
     (forall (?r (relation ?r))
         (<=> (functional ?r)
              (forall (?x ((collection1 ?r) ?x)
                       ?y1  ((collection2 ?r) ?y1)
                       ?y2  ((collection2 ?r) ?y2))
                  (=> (and (?r ?x ?y1) (?r ?x ?y2))
                      (= ?y1 ?y2)))))

(52) (collection total-functional)
     (subcollection total-functional relation)
     (= total-functional (binary-intersection [total functional]))
```

Functions

- A KIF function represents the notion of a map, a so-called "black-box," or an input-output device. A *partial* KIF *function* has three component collections: *source*, *target* and *domain* (of definition). Each of these concepts is represented by a (total) function whose source is the *partial function* collection and whose target is the *collection* collection. We use the notation $f : X \rightarrow Y$ to indicate the source-target typing of a partial KIF function. Most functions in application are total.

$$X \xrightarrow{f} Y$$

Figure 2: Function

```
(53) (function source)
     (= (source source) partial-function)
     (= (target source) collection)

(54) (function target)
     (= (source target) partial-function)
     (= (target target) collection)

(55) (function domain)
     (= (source domain) partial-function)
     (= (target domain) collection)

(56) (forall (?f (partial-function ?f))
         (and (subcollection (domain ?f) (source ?f))
              (forall (?x ?y)
                  (=> (= (?f ?x) ?y)
                      (and ((domain ?f) ?x)
                           ((target ?f) ?y))))))
```

- A partial KIF function should be functional (single valued) – it satisfies the condition: if the function maps an element x in the source to two elements y_1 and y_2 in the target, then the two are one, $y_1 = y_2$.

```
(57) (forall (?f (partial-function ?f)
              ?x ((source ?f) ?x)
              ?y1 ((target ?f) ?y1)
              ?y2 ((target ?f) ?y2))
         (=> (and (= (?f ?x) ?y1)
                  (= (?f ?x) ?y2))
             (= ?y1 ?y2)))
```

- In addition, a partial KIF function should only be defined on its domain (of definition) and it should be total there.

```
(58) (forall (?f (partial-function ?f)
              ?x ((source ?f) ?x))
         (<=> (exists (?y ((target ?f) ?y))
                  (= (?f ?x) ?y))
              ((domain ?f) ?x)))
```

- A (total) KIF *function* is defined. Since its domain equals its source, we can ignore the domain.

```
(59) (forall (?f (function ?f))
         (= (domain ?f) (source ?f)))
```

- A function $f_1 : C_1 \rightarrow D_1$ is a *restriction* of a function $f_2 : C_2 \rightarrow D_2$ when the source (target) of f_1 is a subcollection of the source (target) of f_2 and the functions agree (on source elements of f_1); that is, the functions commute (Diagram 1) with the domain/target inclusions. Of course, we do not have inclusion maps, so we express this pointwise. Restriction is a constraint on the larger function – it says that the larger function maps the source collection of the smaller function into the target collection of the smaller function.

$$\begin{tikzcd} C_1 \arrow[r, hook] \arrow[d, "f_1"'] & C_2 \arrow[d, "f_2"] \\ D_1 \arrow[r, hook] & D_2 \end{tikzcd}$$

Diagram 1: Restriction

```
(60) (relation restriction)
     (= (collection1 restriction) function)
     (= (collection2 restriction) function)
     (forall (?f1 (function ?f1) ?f2 (function ?f2))
         (<=> (restriction ?f1 ?f2)
              (and (subcollection (source ?f1) (source ?f2))
```

```
                   (subcollection (target ?f1) (target ?f2))
                   (forall (?x ((source ?f1) ?x))
                       (= (?f1 ?x) (?f2 ?x))))))
```

- When restricted to its domain, a partial function becomes a total function.

```
(61) (function domain-restriction)
     (= (source domain-restriction) partial-function)
     (= (target domain-restriction) function)
     (forall (?f (partial-function ?f))
         (and (= (source (domain-restriction ?f)) (domain ?f))
              (= (target (domain-restriction ?f)) (target ?f))
              (forall (?x ((domain ?f) ?x))
                  (= ((domain-restriction ?f) ?x) (?f ?x)))))
```

- A partial function $f_1 : C_1 \rightarrow D_1$ is a *partial-restriction* of a partial function $f_2 : C_2 \rightarrow D_2$ when the domain restriction of f_1 is a restriction of the domain restriction of f_2.

```
(62) (relation partial-restriction)
     (= (collection1 partial-restriction) partial-function)
     (= (collection2 partial-restriction) partial-function)
     (forall (?f1 (partial-function ?f1) ?f2 (partial-function ?f2))
         (<=> (partial-restriction ?f1 ?f2)
              (restriction (domain-restriction ?f1) (domain-restriction ?f2))))
```

- The domain restriction of a partial function is a partial restriction of itself.

```
(63) (forall (?f (partial-function ?f))
         (partial-restriction (domain-restriction ?f) ?f))
```

- The domain restriction of a (total) function f is itself.

```
(64) (forall (?f (function ?f))
         (= (domain-restriction ?f) ?f))
```

- One total function is a restriction of another total function <u>iff</u> the relation associated with the first function is an abridgment of the relation associated with the second functions.

```
(65) (forall (?f1 (function ?f1) ?f2 (function ?f2))
         (<=> (restriction ?f1 ?f2)
              (abridgment (fn2rel ?f1) (fn2rel ?f2))))
```

- For any collection C there is an *identity* function.

```
(66) (function identity)
     (= (source identity) collection)
     (= (target identity) function)
     (forall (?c (collection ?c))
         (and (= (source (identity ?c)) ?c)
              (= (target (identity ?c)) ?c)
              (forall (?x (?c ?x))
                  (= ((identity ?c) ?x) ?x))))
```

- The collection $\mathit{function}^2 = \mathit{function} \times \mathit{function}$ of all pairs of functions is defined.

```
(67) (collection pair-function}
     (forall (?f)
         (<=> (pair-function ?f)
              (and (pair ?f)
                   (function (?f 1))
                   (function (?f 2)))))
```

- A *tuple of functions* is a tuple whose target is *function*.

```
(68) (collection tuple-function)
     (subcollection tuple-function tuple)
     (forall (?f (tuple ?f))
         (<=> (tuple-function ?f)
              (= (target ?f) function)))
```

- The *tuple-source* (*tuple-target*) function maps a function tuple $f_n : A_n \rightarrow B_n$ to its source (target) class tuple A_n (B_n).

```
(69) (function tuple-source)
```

```
     (= (source tuple-source) tuple-function)
     (= (target tuple-source) tuple-collection)
     (forall (?f (tuple-function ?f))
         (and (= (arity (tuple-source ?f)) (arity ?f))
              (forall (?n ((arity ?f) ?n))
                  (= ((tuple-source ?f) ?n) (source (?f ?n))))))

(70) (function tuple-target)
     (= (source tuple-target) tuple-function)
     (= (target tuple-target) tuple-collection)
     (forall (?f (tuple-function ?f))
         (and (= (arity (tuple-target ?f)) (arity ?f))
              (forall (?n ((arity ?f) ?n))
                  (= ((tuple-target ?f) ?n) (target (?f ?n))))))
```

- For any function $f : A \rightarrow B$, and any element $y \in B$, the *fiber* of y along f is the collection $f^{-1}(y) = \{x \in A \mid f(x) = y\} \subseteq A$.

```
(71) (KIF$function fiber)
     (KIF$source fiber function)
     (KIF$target fiber function)
     (forall (?f (function ?f))
         (and (= (source (fiber ?f)) (target ?f))
              (forall (?y ((target ?f) ?y))
                  (and (subcollection ((fiber ?f) ?y) (source ?f))
                       (forall ?x ((source ?f) ?x))
                           (<=> (((fiber ?f) ?y) ?x)
                                (= (?f ?x) ?y)))))))
```

- For any two collections C and D and any element $y \in D$ there is a *constant y* function from C to D.

```
(72) (function constant)
     (= (source constant) pair-collection)
     (= (target constant) function)
     (forall (?c (collection ?c) ?d (collection ?d))
         (and (= (source (constant [?c ?d])) ?d)
              (= (target (constant [?c ?d])) function)
              (forall (?y (?d ?y))
                  (and (= (source ((constant [?c ?d]) ?y)) ?c)
                       (= (target ((constant [?c ?d]) ?y)) ?d)
                       (forall (?x (?c ?x))
                           (= (((constant [?c ?d]) ?y) ?x) ?y))))))
```

- For any two collections that are ordered by inclusion $A \subseteq B$, there is an *inclusion* KIF function $A \rightarrow B$.

```
(73) (function inclusion)
     (= (source inclusion) sub-collection)
     (= (target inclusion) function)
     (forall (?a (collection ?a) ?b (collection ?b) (subcollection ?a ?b))
         (and (= (source (inclusion [?a ?b])) ?a)
              (= (target (inclusion [?a ?b])) ?b)
              (forall (?x (?a ?x))
                  (= ((inclusion [?a ?b]) ?x) ?x))))
```

- A KIF function is an *injection* when no distinct source elements have the same image.

```
(74) (collection injection)
     (subcollection injection function)
     (forall (?f (function ?f))
         (<=> (injection ?f)
              (forall (?x1 ((source ?f) ?x1)
                       ?x2 ((source ?f) ?x2))
                  (=> (= (?f ?x1) (?f ?x2))
                      (= ?x1 ?x2)))))
```

- A KIF function is a *surjection* when all elements of the target class are images.

```
(75) (collection surjection)
     (subcollection surjection function)
     (forall (?f (function ?f))
         (<=> (surjection ?f)
              (forall (?y ((target ?f) ?y))
```

```
                  (exists (?x ((source ?f) ?x))
                      (= (?f ?x) ?y)))))
```

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

```
(76) (collection bijection)
     (subcollection bijection function)
     (= bijection (binary-intersection [injection surjection]))
```

- Any partial KIF function can be mapped to a functional KIF relation.

```
(77) (function pfn2rel)
     (= (source pfn2rel) partial-function)
     (= (target pfn2rel) relation)
     (forall (?f (partial-function ?f))
         (and (= (collection1 (pfn2rel ?f)) (source ?f))
              (= (collection2 (pfn2rel ?f)) (target ?f))
              (forall (?x ((source ?f) ?x) ?y ((target ?f) ?y))
                  (<=> ((pfn2rel ?f) ?x ?y)
                       (= (?f ?x) ?y)))))
```

- The *pfn2rel* function can be restricted (at the target) to a bijection from *partial-function* to *functional*. As a result, partial functions and functional relations are isomorphic.

```
(78) (isomorphic partial-function functional)
```

- Any (total) KIF function can be mapped to a total functional KIF relation.

```
(79) (function fn2rel)
     (= (source fn2rel) function)
     (= (target fn2rel) relation)
     (forall (?f (function ?f))
         (and (= (collection1 (fn2rel ?f)) (source ?f))
              (= (collection2 (fn2rel ?f)) (target ?f))
              (forall ?x ((source ?f) ?x) ?y ((target ?f) ?y))
                  (<=> ((fn2rel ?f) ?x ?y)
                       (= (?f ?x) ?y)))))
```

- Clearly, the *pfn2rel* function agrees with the *fn2rel* function on (total) functions.

```
(80) (restriction fn2rel pfn2rel)
```

- The *fn2rel* function can be restricted (at the target) to a bijection from *function* to *total-functional*. As a result, functions and total functional relations are isomorphic.

```
(81) (isomorphic function total-functional)
```

Limits & Colimits

- The unit collection is a "terminal" collection – for any collection *C*, there is exactly one function from *C* to *unit*.

```
(82) (forall (?c (collection ?c))
         (exists (?u (function ?u))
             (and (= (source ?u) ?c)
                  (= (target ?u) unit)
                  (forall (?f (function ?f))
                      (=> (and (= (source ?f) ?c)
                               (= (target ?f) unit))
                          (= ?f ?u))))))
```

- This function is named the *unique* function from *C* to *unit*.

```
(83) (function unique)
     (= (source unique) collection)
     (= (target unique) function)
     (forall (?c (collection ?c))
         (and (= (source (unique ?c)) ?c)
              (= (target (unique ?c)) unit)
              (forall (?x (?c ?x))
                  (= ((unique ?c) ?x) 1))))
```

- The null collection is an "initial" collection – for any collection *C*, there is exactly one function from *null* to *C*.

```
(84) (forall (?c (collection ?c))
         (exists (?u (function ?u))
             (and (= (source ?u) null)
                  (= (target ?u) ?c)
                  (forall (?f (function ?f))
                      (=> (and (= (source ?f) null)
                               (= (target ?f) ?c))
                          (= ?f ?u))))))
```

- This function is named the *counique* (or *empty*) function from *null* to *C*.

```
(85) (function counique)
     (= (source counique) collection)
     (= (target counique) function)
     (forall (?c (collection ?c))
         (and (= (source (counique ?c)) null)
              (= (target (counique ?c)) ?c)))
```

- The *binary Cartesian product* of two arbitrary collections is defined. This is an abbreviated form of the binary product in the quasi-category of collections and their (total) functions – abbreviated since no product projection maps are introduced.

```
(86) (function binary-product)
     (= (source binary-product) pair-collection)
     (= (target binary-product) collection)
     (forall (?c1 (collection ?c1) ?c2 (collection ?c2) ?x)
         (<=> ((binary-product [?c1 ?c2]) ?x)
              (and (pair ?x) (?c1 (?x 1)) (?c2 (?x 2)))))
```

- The *span* collection consists of pairs of functions with common source collection.

```
(87) (collection span)
     (subcollection span pair-function)
     (forall (?f1 (function ?f1) ?f2 (function ?f2))
         (<=> (span [?f1 ?f2])
              (= (source ?f1) (source ?f2))))
```

- For any two functions $f_1 : A \rightarrow C_1$ and $f_2 : A \rightarrow C_2$ with common source collection (that is, for any span) there is a *pairing* function $\langle f_1, f_2\rangle : A \rightarrow C_1 \times C_2$, which pairs the images of the original two functions. This corresponds to the pairing convenience term associated with the mediator of class pairs.

```
(88) (function pairing)
     (= (source pairing) span)
     (= (target pairing) function)
```

```
(forall (?f1 ?f2 (span [?f1 ?f2]))
    (and (= (source (pairing [?f1 ?f2])) (source ?f1))
         (= (target (pairing [?f1 ?f2]))
            (binary-product [(target ?f1) (target ?f2)]))
         (forall (?x ((source (pairing [?f1 ?f2])) ?x))
             (= ((pairing [?f1 ?f2]) ?x) [(?f1 ?x) (?f2 ?x)]))))
```

- The *binary coproduct* (*binary sum*) of two arbitrary collections is defined.

```
(89) (function binary-coproduct)
     (function binary-sum)
     (= binary-coproduct binary-sum)
     (= (source binary-coproduct) pair-collection)
     (= (target binary-coproduct) collection)
     (forall (?c1 (collection ?c1) ?c2 (collection ?c2) ?x)
         (<=> ((binary-coproduct [?c1 ?c2]) ?z)
              (or (exists (?x1 (?c1 ?x1)) (= ?z [1 ?x1]))
                  (exists (?x2 (?c2 ?x2)) (= ?z [2 ?x2])))))
```

- The *opspan* collection consists of pairs of functions with common target collection.

```
(90) (collection opspan)
     (subcollection opspan pair-function)
     (forall (?g1 (function ?g1) ?g2 (function ?g2))
         (<=> (opspan [?g1 ?g2])
              (= (target ?g1) (target ?g2))))
```

- For any two functions $f_1 : C_1 \rightarrow B$ and $f_2 : C_2 \rightarrow B$ with common target collection (that is, for any opspan) there is a *copairing* function $[f_1, f_2] : C_1 + C_2 \rightarrow B$, which is equal to the original two functions on their domains. This corresponds to the copairing convenience term associated with the comediator of class pairs. Any function can be built from scratch using multiple copairing of constant functions. The definition is represented via the recommended "guarded command" expression for disjoint unions.

```
(91) (function copairing)
     (= (source copairing) opspan)
     (= (target copairing) function)
     (forall (?f1 ?f2 (opspan [?f1 ?f2]))
         (and (= (source (copairing [?f1 ?f2]))
                 (binary-coproduct [(source ?f1) (source ?f2)]))
              (= (target (copairing [?f1 ?f2])) (target ?f1))
              (forall (?z ((binary-coproduct [(source ?f1) (source ?f2)]) ?z))
                  (or (=> (exists (?x1 ((source ?f1) ?x1)) (= ?z [1 ?x1]))
                          (= ((copairing [?f1 ?f2]) ?z) (?f1 ?x1)))
                      (=> (exists (?x2 ((source ?f2) ?x2)) (= ?z [2 ?x2]))
                          (= ((copairing [?f1 ?f2]) ?z) (?f2 ?x2)))))))
```

- The *ternary Cartesian product* of three arbitrary collections is defined.

```
(92) (function ternary-product)
     (= (source ternary-product) triple-collection)
     (= (target ternary-product) collection)
     (forall (?c1 (collection ?c1) ?c2 (collection ?c2) ?c3 (collection ?c3) ?x)
         (<=> ((ternary-product [?c1 ?c2 ?c3]) ?x)
              (and (triple ?x) (?c1 (?x 1)) (?c2 (?x 2)) (?c3 (?x 3)))))
```

- The Cartesian *product* $\prod C$ of a tuple collection C is defined.

```
(93) (function product)
     (= (source product) tuple-collection)
     (= (target product) collection)
     (forall (?c (tuple-collection ?c) ?t)
         (<=> ((product ?c) ?t)
              (and (tuple ?t)
                   (= (arity ?t) (arity ?c))
                   (forall (?n ((source ?c) ?n))
                       ((?c ?n) (?t ?n))))))
```

- Binary product is a restriction of arbitrary product.

```
(94) (restriction binary-product product)
```

- For any tuple of functions $f_n : A \rightarrow C_n$ with common source collection, there is a *tupling* function $\langle f_n \rangle : A \rightarrow \prod_n C_n$ that tuples the images of the component functions. Tupling is parameterized by the target tuple collection.

```
(95) (function tupling)
     (= (source tupling) tuple-collection)
     (= (target tupling) partial-function)
     (forall (?c (tuple-collection ?c))
         (and (= (source (tupling ?c)) (power [(arity ?c) function]))
              (= (target (tupling ?c)) function)
              (forall (?f ((power [(arity ?c) function]) ?f))
                  (<=> ((domain (tupling ?c)) ?f)
                       (and (forall (?j ((arity ?c) ?j) ?k ((arity ?c) ?k))
                                (= (source (?f ?j)) (source (?f ?k))))
                            (forall (?n ((arity ?c) ?n))
                                (= (target (?f ?n)) (?c ?n))))))))
     (forall (?c (tuple-collection ?c)
              ?f ((domain (tupling ?c)) ?f))
         (and (= (target ((tupling ?c) ?f)) (product ?c))
              (forall (?n ((arity ?c) ?n))
                  (and (= (source ((tupling ?c) ?f)) (source (?f ?n)))
                       (forall (?x ((source (?f ?n)) ?x))
                           (= ((((tupling ?c) ?f) ?x) ?n) ((?f ?n) ?x)))))))
```

- The *coproduct* (*sum* or *disjoint union*) $\sum C$ of a tuple collection C is defined.

```
(96) (function coproduct)
     (function sum)
     (function disjoint-union)
     (= coproduct sum)
     (= sum disjoint-union)
     (= (source coproduct) tuple-collection)
     (= (target coproduct) collection)
     (forall (?c (tuple-collection ?c) ?z)
         (<=> ((coproduct ?c) ?z)
              (and (pair ?z)
                   (exists (?n ((arity ?c) ?n))
                       (and (= (?z 1) ?n) ((?c ?n) (?z 2)))))))
```

- Binary coproduct is a restriction of arbitrary coproduct.

```
(97) (restriction binary-coproduct coproduct)
```

- For any tuple of functions $f_n : C_n \rightarrow B$ with common target collection, there is a *cotupling* function $[f_n] : \sum_n C_n \rightarrow B$ that independently applies the component functions. Cotupling is parameterized by the source tuple collection.

```
(98) (function cotupling)
     (= (source cotupling) tuple-collection)
     (= (target cotupling) partial-function)
     (forall (?c (tuple-collection ?c))
         (and (= (source (cotupling ?c)) (power [(arity ?c) function]))
              (= (target (cotupling ?c)) function)
              (forall (?f ((power [(arity ?c) function]) ?f))
                  (<=> ((domain (cotupling ?c)) ?f)
                       (and (forall (?j ((arity ?c) ?j) ?k ((arity ?c) ?k))
                                (= (target (?f ?j)) (target (?f ?k))))
                            (forall (?n ((arity ?c) ?n))
                                (= (source (?f ?n)) (?c ?n))))))))
     (forall (?c (tuple-collection ?c)
              ?f ((domain (cotupling ?c)) ?f))
         (and (= (source ((cotupling ?c) ?f)) (coproduct ?c))
              (forall (?n ((arity ?c) ?n))
                  (and (= (target ((cotupling ?c) ?f)) (target (?f ?n)))
                       (forall (?x ((source (?f ?n)) ?x))
                           (= (((cotupling ?c) ?f) [?n ?x]) ((?f ?n) ?x)))))))
```

- The Cartesian *power* (or *exponent* C^J or *hom-collection* $[J, C]$) of a base collection C with respect to an index collection J is the Cartesian product, where all of the factors are the collection C. Note: the Cartesian power is <u>not</u> the subcollection power $\wp C$.

```
(99) (function power)
     (function exponent)
     (= power exponent)
     (= (source power) pair-collection)
     (= (target power) collection)
     (forall (?j (collection ?j) ?c (collection ?c) ?t)
         (<=> ((power [?j ?c]) ?t)
              (and (function ?t)
                   (= (source ?t) ?j)
                   (= (target ?t) ?c))))
```

○ The *pullback* of opspans is defined.

```
(100) (function pullback)
      (= (source pullback) opspan)
      (= (target pullback) collection)
      (forall (?s (opspan ?s))
          (and (subcollection
                   (pullback ?s)
                   (binary-product [(source (?s 1)) (source (?s 2))]))
               (<=> ((pullback ?s) ?x)
                    (= ((?s 1) (?x 1)) ((?s 2) (?x 2))))))
```

o For any pair of functions $f_1 : A \rightarrow C_1$ and $f_2 : A \rightarrow C_2$ with common source collection, there is a *pullback-pairing* function $\langle f_1, f_2 \rangle : A \rightarrow C_1 \times_C C_2$ that pairs the images of the component functions. Pairing is parameterized by an opspan under the target collection pair that commutes with the component functions.

```
(101) (function pullback-pairing)
      (= (source pullback-pairing) opspan)
      (= (target pullback-pairing) partial-function)
      (forall (?s (opspan ?s))
          (and (= (source (pullback-pairing ?s)) pair-function)
               (= (target (pullback-pairing ?s)) function)
               (forall (?f1 ?f2 (pair-function [?f1 ?f2]))
                   (<=> ((domain (pullback-pairing ?s)) [?f1 ?f2])
                        (and (= (source ?f1) (source ?f2))
                             (= (target ?f1) (source (?s 1)))
                             (= (target ?f2) (source (?s 2)))
                             (forall (?x ((source ?f1) ?x))
                                 (= ((?s 1) (?f1 ?x)) ((?s 2) (?f2 ?x)))))))))
      (forall (?s (opspan ?s)
               ?f1 ?f2 ((domain (pullback-pairing ?s)) [?f1 ?f2]))
          (and (= (target ((pullback-pairing ?c) [?f1 ?f2]))
                  (binary-product [(source (?s 1)) (source (?s 2))]))
               (= (source ((pullback-pairing ?c) [?f1 ?f2])) (source ?f1))
               (forall (?x ((source ?f1) ?x))
                   (= (((pullback-pairing ?s) [?f1 ?f2]) ?x) [(?f1 ?x) (?f2 ?x)]))))
```