# The IFF Classification Ontology

This document declares and axiomatizes the IFF Classification Ontology. The IFF Classification Ontology provides a formalism for the theory of *Distributed Conceptual Structures* (Kent 2001). This theory is concerned with the distribution and conception of knowledge. It rests upon two related theories, Information Flow and Formal Concept Analysis, which it seeks to unify. Information Flow (IF) (Barwise and Seligman 1997) is concerned with the distribution of knowledge. The foundations of Information Flow is explicitly based upon a mathematical theory known as the Chu Construction in *-autonomous categories (Barr 1991). Formal Concept Analysis (FCA) (Ganter and Wille 1999) is concerned with the conception and analysis of knowledge. The theory of distributed conceptual structures merges these two studies by categorizing the basic theorem of Formal Concept Analysis, thus extending it to the distributed realm of Information Flow. The main result of the merged theory is the representation of the basic theorem of Formal Concept Analysis as the categorical equivalence between classifications and concept lattices at the level of functions, and the categorical equivalence between bonds and the opposite of complete adjoints and between bonding pairs and complete homomorphisms at the level of relations. This accomplishes a rapprochement between Information Flow and Formal Concept Analysis. The IFF Classification Ontology currently contains 286 non-identical terms (288 terms with 2 synonyms), partitioned into 32 terms for large concept lattices (`CL`, `CL.MOR`), 41 terms for large complete lattices (`LAT`, `LAT.ADJ`, `LAT.MOR`), and 213 terms for large classifications (`CLS`, `CLS.CL`, `CLS.FIB`, `CLS.COLL`, `CLS.INFO`, `CLS.REL`, `CLS.BND`, `CLS.BNDPR`, `CLS.COL`, `CLS.COL.COPRD`, `CLS.COL.COINV`, `CLS.COL.COEQ`, `CLS.COL.PSH`).

## The Namespace of Large Concept Lattices

This namespace will represent large concept lattices and their morphisms. The terms introduced in this namespace are listed in Table 1.

**Table 1: Terms introduced into the large concept lattice namespace**

| | Collection | Function | Other |
|---|---|---|---|
| `CL` | `concept-lattice` | `complete-lattice = underlying`<br>`instance type`<br>`instance-embedding type-embedding`<br>`partial-order class`<br>`classification`<br>`opposite instance-power` | |
| `CL .MOR` | `concept-morphism` | `source target adjoint-pair`<br>`instance type`<br>`infomorphism`<br>`instance-join type-meet right-representation`<br>`extent intent left-representation`<br>`representation`<br>`opposite composition identity`<br>`instance-power` | `composable-opspan`<br>`composable` |

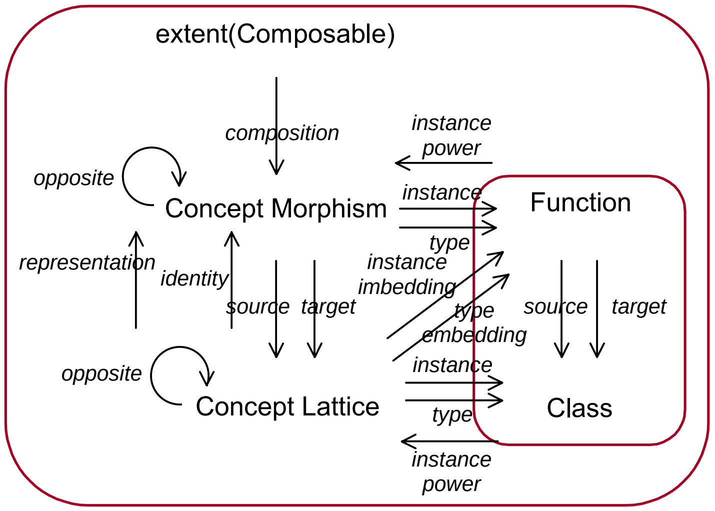


**Diagram 1: Core Collections and Functions for Concept Lattices**

## Concept Lattices

**CL**

- An (abstract) *concept lattice* $\boldsymbol{L} = \langle \mathit{lat}(\boldsymbol{L}), \mathit{inst}(\boldsymbol{L}), \mathit{typ}(\boldsymbol{L}), \iota_{\boldsymbol{L}}, \tau_{\boldsymbol{L}} \rangle$ (Figure 1) consists of a complete lattice $\mathit{lat}(\boldsymbol{L})$, two classes $\mathit{inst}(\boldsymbol{L})$ and $\mathit{typ}(\boldsymbol{L})$ called the instance class and the type class of $\boldsymbol{L}$, respectively; along with two functions, an instance embedding function $\iota_{\boldsymbol{L}} : \mathit{inst}(\boldsymbol{L}) \rightarrow \mathit{lat}(\boldsymbol{L})$ and a type embedding function $\tau_{\boldsymbol{L}} : \mathit{typ}(\boldsymbol{L}) \rightarrow \mathit{lat}(\boldsymbol{L})$, which satisfying the following conditions.
  - The image $\iota_{\boldsymbol{L}}(\mathit{inst}(\boldsymbol{L}))$ is join-dense in $\mathit{lat}(\boldsymbol{L})$.
  - The image $\tau_{\boldsymbol{L}}(\mathit{typ}(\boldsymbol{L}))$ is meet-dense in $\mathit{lat}(\boldsymbol{L})$.

```
(1) (KIF$collection concept-lattice)

(2) (KIF$function complete-lattice)
    (KIF$function underlying)
    (= underlying complete-lattice)
    (= (KIF$source complete-lattice) concept-lattice)
    (= (KIF$target complete-lattice) LAT$complete-lattice)

(3) (KIF$function instance)
    (= (KIF$source instance) concept-lattice)
    (= (KIF$target instance) SET$class)

(4) (KIF$function type)
    (= (KIF$source type) concept-lattice)
    (= (KIF$target type) SET$class)

(5) (KIF$function instance-embedding)
    (= (KIF$source instance-embedding) concept-lattice)
    (= (KIF$target instance-embedding) SET.FTN$function)
    (forall (?l (concept-lattice ?l))
        (and (= (SET.FTN$source (instance-embedding ?l)) (instance ?l))
             (= (SET.FTN$target (instance-embedding ?l))
                (ORD$class (LAT$partial-order (complete-lattice ?l))))))

(5) (KIF$function type-embedding)
    (= (KIF$source type-embedding) concept-lattice)
    (= (KIF$target type-embedding) SET.FTN$function)
    (forall (?l (concept-lattice ?l))
        (and (= (SET.FTN$source (type-embedding ?l)) (type ?l))
             (= (SET.FTN$target (type-embedding ?l))
                (ORD$class (LAT$partial-order (complete-lattice ?l))))))

(6) (forall (?a (CLS$classification ?a))
        (and ((ORD$join-dense (LAT$partial-order (complete-lattice ?l)))
                 (SET.FTN$image (instance-embedding ?l)))
             ((ORD$meet-dense (LAT$partial-order (complete-lattice ?l)))
                 (SET.FTN$image (type-embedding ?l)))))
```

$$\begin{tikzcd}
\mathit{typ}(\boldsymbol{L}) \arrow[d, "\tau_{\boldsymbol{L}}"] \\
\mathit{lat}(\boldsymbol{L}) \arrow[d, dash, "\leq_{\boldsymbol{L}}"] \\
\mathit{lat}(\boldsymbol{L}) \\
\mathit{inst}(\boldsymbol{L}) \arrow[u, "\iota_{\boldsymbol{L}}"']
\end{tikzcd}$$

**Figure 1: Concept Lattice**

- Here we define two convenience terms – the *partial order* underlying a concept lattice and the *class* of elements in the concept lattice. These facilitate the expression of some of the axioms below.

```
(7) (KIF$function partial-order)
    (= (KIF$source partial-order) concept-lattice)
    (= (KIF$target partial-order) ORD$partial-order)
    (forall (?l (concept-lattice ?l))
        (= (partial-order ?l) (LAT$partial-order (complete-lattice ?l))))

(8) (KIF$function class)
    (= (KIF$source class) concept-lattice)
    (= (KIF$target class) SET$class)
    (forall (?l (concept-lattice ?l))
        (= (class ?l) (ORD$class (partial-order ?l))))
```

- Any concept lattice $\boldsymbol{L} = \langle \mathit{lat}(\boldsymbol{L}), \mathit{inst}(\boldsymbol{L}), \mathit{typ}(\boldsymbol{L}), \iota_{\boldsymbol{L}}, \tau_{\boldsymbol{L}} \rangle$ has an associated *classification* $\boldsymbol{A} = \langle \mathit{inst}(\boldsymbol{A}), \mathit{typ}(\boldsymbol{A}), \vDash_{\boldsymbol{A}} \rangle$ whose incidence relation is defined by $i \vDash_{\boldsymbol{A}} t$ iff $\iota(i) \leq_{\boldsymbol{A}} \tau(t)$.

> The classification associated with a concept lattice is the image of the object function of the classification functor applied to the concept lattice:
>
> $C$ : CONCEPT LATTICE $\rightarrow$ CLASSIFICATION.

```
(9) (KIF$function classification)
    (= (KIF$source classification) concept-lattice)
    (= (KIF$target classification) CLS$classification)
    (forall (?l (concept-lattice ?l))
        (and (= (CLS$instance (classification ?l)) (instance ?l))
             (= (CLS$type (classification ?l)) (type ?l))
             (forall (?i ((instance ?l) ?i) ?t ((type ?l) ?t))
                 (<=> ((classification ?l) ?i ?t)
                      ((partial-order ?l)
                          ((instance-embedding ?l) ?i) ((type-embedding ?l) ?t))))))
```

- From properties discussed above, it can be immediately proven that the composition of ‘`concept-lattice`’ and ‘`classification`’ is the identity on the ‘`classification`’ `collection`. We state this in an external namespace.

```
(forall (?c (CLS$classification ?c))
    (= (CL$classification (CLS.CL$concept-lattice ?c)) ?c))
```

- Also, from properties discussed above, it can be immediately proven that for any complete lattice L, the complete lattice of the concept lattice of $L$ is $L$ itself; that is, that the composition of ‘`concept-lattice`’ and ‘`complete-lattice`’ is the identity on the ‘`complete-lattice`’ `collection`. We state this in an external namespace.

```
(forall (?l (LAT$complete-lattice ?l))
    (= (CL$complete-lattice (LAT$concept-lattice ?l)) ?l))
```

- For any concept lattice $\boldsymbol{L} = \langle \mathit{lat}(\boldsymbol{L}), \mathit{inst}(\boldsymbol{L}), \mathit{typ}(\boldsymbol{L}), \iota_{\boldsymbol{L}}, \tau_{\boldsymbol{L}} \rangle$, the *opposite* or *dual* of $\boldsymbol{L}$ is the concept lattice $\boldsymbol{L}^{\perp} = \langle \mathit{lat}(\boldsymbol{L})^{\perp}, \mathit{typ}(\boldsymbol{L}), \mathit{inst}(\boldsymbol{L}), \tau_{\boldsymbol{L}}, \iota_{\boldsymbol{L}} \rangle$, whose instances are the types of $\boldsymbol{L}$, whose types are the instances of $\boldsymbol{L}$, whose instance embedding function is the type embedding function of $\boldsymbol{L}$, whose type embedding function is the instance embedding function of $\boldsymbol{L}$, and whose complete lattice is the opposite of the complete lattice of $\boldsymbol{L}$ (turn it upside down). Axiom (7) specifies the opposite operator on concept lattices.

```
(10) (KIF$function opposite)
     (= (KIF$source opposite) concept-lattice)
     (= (KIF$target opposite) concept-lattice)
     (forall (?l (concept-lattice ?l))
         (and (= (complete-lattice (opposite ?l))
                 (LAT$opposite (complete-lattice ?l)))
              (= (instance (opposite ?l)) (type ?l))
              (= (type (opposite ?l)) (instance ?l))
              (= (instance-embedding (opposite ?l)) (type-embedding ?l))
              (= (type-embedding (opposite ?l)) (instance-embedding ?l))))
```

- For any class $A$ the *instance power concept lattice* $\wp A = \langle\langle \wp A, \subseteq_A, \cap_A, \cup_A \rangle, A, \wp A, \{\text{-}\}_A, id_{\wp A} \rangle$ over $A$ is defined as follows: the complete lattice is the power lattice generated by $A$, the instance class is $A$; the type class is the power class $\wp A$ (so that a type is a subclass of $A$), the instance embedding function is the singleton function for $A$, and the type-embedding function is the identity. Axiom (8) specifies the instance power operator from classes to concept lattices.

```
(11) (KIF$function instance-power)
     (= (KIF$source instance-power) SET$class)
     (= (KIF$target instance-power) concept-lattice)
     (forall (?c (SET$class ?c))
         (and (= (complete-lattice (instance-power ?c)) (LAT$power ?c))
              (= (instance (instance-power ?c)) ?c)
              (= (type (instance-power ?c)) (SET$power ?c))
              (= (instance-embedding (instance-power ?c)) (SET.FTN$singleton ?c))
              (= (type-embedding (instance-power ?c))
                 (SET.FTN$identity (SET$power ?c)))))
```

## Concept Morphisms

**CL.MOR**

$$\begin{tikzcd}
typ(\boldsymbol{L}) \arrow[r, "typ(\boldsymbol{f})"] \arrow[d, "\tau_{\boldsymbol{L}}"] & typ(\boldsymbol{K}) \arrow[d, "\tau_{\boldsymbol{K}}"] \\
\boldsymbol{L} \arrow[r, "right(adj(\boldsymbol{f}))"] \arrow[d, dash, "\leq_{\boldsymbol{L}}"] & \boldsymbol{K} \arrow[d, dash, "\leq_{\boldsymbol{K}}"] \\
\boldsymbol{L} & \boldsymbol{K} \arrow[l, "left(adj(\boldsymbol{f}))"] \\
inst(\boldsymbol{L}) \arrow[u, "\iota_{\boldsymbol{L}}"] & inst(\boldsymbol{K}) \arrow[l, "inst(\boldsymbol{f})"] \arrow[u, "\iota_{\boldsymbol{K}}"]
\end{tikzcd}$$

**Figure 2: Concept Lattice Morphism**

o Concept lattices are related through concept morphisms. An (abstract) *concept morphism* $\boldsymbol{f} : \boldsymbol{L} \rightleftarrows \boldsymbol{K}$ from (abstract) concept lattice $\boldsymbol{L}$ to (abstract) concept lattice $\boldsymbol{K}$ (Figure 2) consists of a pair of oppositely directed functions, $inst(\boldsymbol{f}) : inst(\boldsymbol{K}) \rightarrow inst(\boldsymbol{L})$ and $typ(\boldsymbol{f}) : typ(\boldsymbol{L}) \rightarrow typ(\boldsymbol{K})$, between instance classes and type classes, and an adjoint pair of monotonic functions $adj(\boldsymbol{f}) : \boldsymbol{K} \rightleftarrows \boldsymbol{L}$, where the right adjoint $right(\boldsymbol{f}) : \boldsymbol{L} \rightarrow \boldsymbol{K}$ is a monotonic function in the forward direction (for the concept lattice morphism, not the adjoint pair) that preserves types (in the sense that the upper rectangle in Figure 5 is commutative)

$$\tau_{\boldsymbol{L}} \cdot right(adj(\boldsymbol{f})) = typ(\boldsymbol{f}) \cdot \tau_{\boldsymbol{K}}$$

and the left adjoint $left(\boldsymbol{f}) : \boldsymbol{K} \rightarrow \boldsymbol{L}$ is a monotonic function in the reverse direction that preserves instances (in the sense that the lower rectangle in Figure 2 is commutative)

$$\iota_{\boldsymbol{K}} \cdot left(adj(\boldsymbol{f})) = inst(\boldsymbol{f}) \cdot \iota_{\boldsymbol{L}} .$$

Note the contravariance between the concept morphism and the adjoint pair – the concept morphism is oriented in the same direction as the type function, whereas the adjoint pair is oriented in the same direction as the left adjoint. Let CONCEPT LATTICE denote the quasi-category of concept lattices and concept morphism.

```
(1) (KIF$collection concept-morphism)

(2) (KIF$function source)
    (= (KIF$source source) concept-morphism)
    (= (KIF$target source) CL$concept-lattice)

(3) (KIF$function target)
    (= (KIF$source target) concept-morphism)
    (= (KIF$target target) CL$concept-lattice)

(4) (KIF$function adjoint-pair)
    (= (KIF$source adjoint-pair concept-morphism)
    (= (KIF$target adjoint-pair) LAT.ADJ$adjoint-pair)
    (forall (?f (concept-morphism ?f))
        (and (= (LAT.ADJ$source (adjoint-pair ?f))
                (CL$complete-lattice (target ?f)))
             (= (LAT.ADJ$target (adjoint-pair ?f))
                (CL$complete-lattice (source ?f)))))

(5) (KIF$function instance)
    (= (KIF$source instance) concept-morphism)
    (= (KIF$target instance) SET.FTN$function)
    (forall (?f (concept-morphism ?f))
        (and (= (SET.FTN$source (instance ?f)) (CL$instance (target ?f)))
             (= (SET.FTN$target (instance ?f)) (CL$instance (source ?f)))
             (= (SET.FTN$composition
                    [(CL$instance-embedding (target ?f))
                     (ORD.FTN$function (LAT.ADJ$left (adjoint-pair ?f)))])
                (SET.FTN$composition
                    [(instance ?f)
                     (CL$instance-embedding (source ?f))]))))

(6) (KIF$function type)
    (= (KIF$source type) concept-morphism)
    (= (KIF$target type) SET.FTN$function)
    (forall (?f (concept-morphism ?f))
        (and (= (SET.FTN$source (type ?f)) (CL$type (source ?f)))
             (= (SET.FTN$target (type ?f)) (CL$type (target ?f)))
             (= (SET.FTN$composition
                    [(CL$type-embedding (source ?f))
                     (ORD.FTN$function (LAT.ADJ$right (adjoint-pair ?f)))])
                (SET.FTN$composition
                    [(type ?f)
                     (CL$type-embedding (target ?f))]))))
```

o Any concept morphism $\boldsymbol{f} : \boldsymbol{L} \rightleftarrows \boldsymbol{K}$ has an associated infomorphism $\mathsf{info}(\boldsymbol{f}) : \mathsf{cls}(\boldsymbol{L}) \rightleftarrows \mathsf{cls}(\boldsymbol{K})$ whose fundamental property, expressed as

$$\mathsf{inst}(\mathsf{info}(\boldsymbol{f}))(i) \vDash_{\mathsf{cls}(\boldsymbol{L})} t \text{ iff } i \vDash_{\mathsf{cls}(\boldsymbol{K})} \mathsf{typ}(\mathsf{info}(\boldsymbol{f}))(t)$$

for all instances $i \in \mathsf{inst}(\boldsymbol{K})$ and all types $t \in \mathsf{typ}(\boldsymbol{L})$, is an easy translation of the adjointness condition for the adjoint pair $\mathsf{adj}(\boldsymbol{f})$ and the commutativity of the instance/type functions with the left/right monotonic functions.

The infomorphism associated with a concept morphism is the image of the morphism function of the classification functor applied to the concept morphism:

$C$ : CONCEPT LATTICE $\rightarrow$ CLASSIFICATION.

```
(7) (KIF$function infomorphism)
    (= (KIF$source infomorphism) concept-morphism)
    (= (KIF$target infomorphism) CLS.INFO$infomorphism)
    (forall (?f (concept-morphism ?l))
        (and (= (CLS.INFO$instance (infomorphism ?f)) (instance ?f))
             (= (CLS.INFO$type (infomorphism ?f)) (type ?f))))
```

o From properties discussed above, it can be immediately proven that the composition of ‘`concept-morphism`’ and ‘`infomorphism`’ is the identity on the ‘`infomorphism`’ `collection`. We state this in an external namespace.

```
    (forall (?f (CLS.INFO$infomorphism ?f))
        (= (CL.MOR$infomorphism (CLS.CL$concept-morphism ?f)) ?f))
```

○ In section 3.1 of the paper (Kent 2001), which is concerned with functional equivalence, the following ideas are introduced and developed in order to demonstrate the categorical equivalence CLASSIFICATION ≡ CONCEPT LATTICE. For any concept lattice $\boldsymbol{L}$, the instance-join function $\mathsf{lat}(\mathsf{cls}(\boldsymbol{L})) \rightarrow \boldsymbol{L}$ maps a formal concept $(A, \Gamma)$ in the complete lattice of the classification of $\boldsymbol{L}$ to the join of the instance-embedding image of its extent. Dually, the type-meet function $\mathsf{lat}(\mathsf{cls}(\boldsymbol{L})) \rightarrow \boldsymbol{L}$ maps a formal concept $(A, \Gamma)$ in the complete lattice of the classification of $\boldsymbol{L}$ to the meet of the type-embedding image of its intent. These two mappings are the same.

```
(8) (KIF$function instance-join)
    (= (KIF$source instance-join) CL$concept-lattice)
    (= (KIF$target instance-join) ORD.FTN$function)
    (forall (?l (CL$concept-lattice ?l))
        (and (= (ORD.FTN$source (instance-join ?l))
                (CL$partial-order (CLS$concept-lattice (CL$classification ?l))))
             (= (ORD.FTN$target (instance-join ?l)) (CL$partial-order ?l))
             (= (ORD.FTN$function (instance-join ?l))
                (SET.FTN$composition
                    [(SET.FTN$composition
                         [(CLS.CL$extent (CL$classification ?l))
                          (SET.FTN$power (CL$instance-embedding ?l))])
                     (LAT$join (CL$complete-lattice ?l))]))))

(9) (KIF$function type-meet)
    (= (KIF$source type-meet) CL$concept-lattice)
    (= (KIF$target type-meet) ORD.FTN$function)
    (forall (?l (CL$concept-lattice ?l))
        (and (= (ORD.FTN$source (type-meet ?l))
                (CL$partial-order (CLS$concept-lattice (CL$classification ?l))))
             (= (ORD.FTN$target (type-meet ?l)) (CL$partial-order ?l))
             (= (ORD.FTN$function (type-meet ?l))
                (SET.FTN$composition
                    [(SET.FTN$composition
                         [(CLS.CL$intent (CL$classification ?l))
                          (SET.FTN$power (CL$type-embedding ?l))])
                     (LAT$meet (CL$complete-lattice ?l))]))))
```

- The previous two functions can be shown to be identical monotonic functions.

```
(forall (?l (CL$concept-lattice ?l))
    (= (instance-join ?l) (type-meet ?l)))
```

- Let us call this common function the *right representation* of $\boldsymbol{L}$.

```
(10) (KIF$function right-representation)
     (= (KIF$source right-representation) CL$concept-lattice)
     (= (KIF$target right-representation) ORD.FTN$function)
     (= right-representation instance-join)
```

- Any concept lattice $\boldsymbol{L}$ is indirectly related to the concept lattice of the classification of $\boldsymbol{L}$ through the following two functions. The *extent* of an element $l \in L$, considered as a concept, is the class of all instances whose generated concept is at or below the concept, $\mathit{extent}(l) = \{a \in \mathit{inst}(\boldsymbol{L}) \mid \iota_{\boldsymbol{L}}(a) \leq_L l\}$. Since the extent of an element $l \in L$, considered as a type in the classification of $\boldsymbol{L}$, is the class $\mathit{extent}_{cls(\boldsymbol{L})}(l) = \{k \in L \mid k \leq_L l\}$, the conceptual extent can expressed as $\mathit{extent}(l) = \iota_{\boldsymbol{L}}^{-1}(\mathit{extent}_{cls(\boldsymbol{L})}(l))$. The *intent* is the dual notion: $\mathit{intent}(l) = \{\alpha \in \mathit{typ}(\boldsymbol{L}) \mid l \leq_L \tau_L(\alpha)\}$. As indicated above, and as we shall axiomatize, both the extent and intent represent concepts of $\boldsymbol{L}$.

```
(11) (KIF$function extent)
     (= (KIF$source extent) CL$concept-lattice)
     (= (KIF$target extent) SET.FTN$function)
     (forall (?l (CL$concept-lattice ?l))
         (and (= (source (extent ?l)) (CL$class ?l))
              (= (target (extent ?l)) (SET$power (CL$instance ?l)))
              (= (extent ?l)
                 (SET.FTN$composition
                     [(CLS$extent (CL$classification ?l))
                      (SET.FTN$inverse-image (CL$instance-embedding ?l))]))))

(12) (KIF$function intent)
     (= (KIF$source intent) CL$concept-lattice)
     (= (KIF$target intent) SET.FTN$function)
     (forall (?l (CL$concept-lattice ?l))
         (and (= (source (intent ?l)) (CL$class ?l))
              (= (target (intent ?l)) (SET$power (CL$type ?l)))
              (= (intent ?l)
                 (SET.FTN$composition
                     [(CLS$intent (CL$classification ?l))
                      (SET.FTN$inverse-image (CL$type-embedding ?l))]))))
```

- The following fact can be proven: there is a unique function, whose source class is the source of the extent and intent functions, whose target is the class underlying the concept lattice of the classification of $\boldsymbol{L}$, whose composition with the extent function of the concept lattice of the classification of $\boldsymbol{L}$ is the above extent function, and whose composition with the intent function of the concept lattice of the classification of $\boldsymbol{L}$ is the above intent function. Moreover, this function preserves order; that is , it is a monotonic function. Let us call this function the *left representation* of $\boldsymbol{L}$. We use a definite description to define this.

```
(13) (KIF$function left-representation)
     (= (KIF$source left-representation) CL$concept-lattice)
     (= (KIF$target left-representation) ORD.FTN$function)
     (forall (?l (CL$concept-lattice ?l))
         (= (left-representation ?l)
            (the (?f (ORD.FTN$function ?f))
              (and (= (source ?f) (CL$partial-order ?l))
                   (= (target ?f)
                      (CL$partial-order (CLS$concept-lattice (CL$classification ?l))))
                   (= (SET.FTN$composition
                          [(ORD.FTN$function ?f)
                           (CLS$extent (CLS$concept-lattice (CL$classification ?l)))])
```

```
                     (extent ?l))
                  (= (SET.FTN$composition
                         [(ORD.FTN$function ?f)
                          (CLS$intent (CLS$concept-lattice (CL$classification ?l)))])
                     (intent ?l))))))
```

- For any concept lattice ***L***, it can be proven that the left and right representation monotonic functions are inverse to each other. This demonstrates that the concept lattice of the classification of ***L*** represents ***L*** via left representation (extent and intent functions).

```
(forall (?l (CL$concept-lattice ?l))
  (and (= (ORD.FTN$composition [(left-representation ?l) (right-representation ?l)])
          (ORD.FTN$identity (CL$class ?l)))
       (= (ORD.FTN$composition [(right-representation ?l) (left-representation ?l)])
          (ORD.FTN$identity
              (CL$class (CLS$concept-lattice (CL$classification ?l)))))))
```

- Since two inverse monotonic functions form an adjoint pair, we can rephrase this in terms of concept morphisms: for any concept lattice ***L***, there is a *representation* concept morphism ***L*** $\rightleftarrows$ *clat*(*cls*(***L***)) from ***L*** to the concept lattice of the classification of ***L***.

For any concept lattice ***L*** the representation concept morphism at ***L*** is the ***L***th component of a natural isomorphism ***L*** ≅ *L*(*C*(***L***)), demonstrating that the quasi-category of concept lattices is categorically equivalent to the quasi-category of classifications:

CINCEPT LATTICE ≡ CLASSIFICATION.

```
(14) (KIF$function representation)
     (= (KIF$source representation) CL$concept-lattice)
     (= (KIF$target representation) concept-morphism)
     (forall (?l (CL$concept-lattice ?l))
         (and (= (source (representation ?l)) ?l)
              (= (target (representation ?l))
                 (CLS$concept-lattice (CL$classification ?l)))
              (= (LAT.ADJ$left (adjoint-pair (representation ?l)))
                 (left-representation ?l))
              (= (LAT.ADJ$right (adjoint-pair (representation ?l)))
                 (right-representation ?l))
              (= (instance (representation ?l))
                 (SET.FTN$identity (CL$instance ?l)))
              (= (type (representation ?l))
                 (SET.FTN$identity (CL$instance ?l)))))
```

- In addition, from properties discussed above, it can be immediately proven that for any (complete lattice) adjoint pair *f*, the adjoint pair of the concept morphism of *f* is *f* itself; that is, that the composition of '`ORD.LAT$concept-morphism`' and '`adjoint-pair`' is the identity on the '`adjoint-pair`' `collection`. We state this in an external namespace.

```
(forall (?f (LAT.ADJ$adjoint-pair ?f))
    (= (CL.MOR$adjoint-pair (LAT.ADJ$concept-morphism ?f)) ?f))
```

Duality can be extended to concept morphisms. For any concept morphism ***f*** : ***L*** $\rightleftarrows$ ***K***, the *opposite* or *dual* of ***f*** is the concept morphism $f^{\perp} : K^{\perp} \rightleftarrows L^{\perp}$, whose source concept lattice is the opposite of the target of ***f***, whose target concept lattice is the opposite of the source of ***f***, whose adjoint pair is the opposite of the adjoint pair of ***f***, whose instance function is the type function of ***f***, whose type function is the instance function of ***f***, and whose preservation conditions have been dualized.

```
(15) (KIF$function opposite)
     (= (KIF$source opposite) concept-morphism)
     (= (KIF$target opposite) concept-morphism)
     (forall (?f (concept-morphism ?f))
         (and (= (source (opposite ?f)) (CL$opposite (target ?f)))
              (= (target (opposite ?f)) (CL$opposite (source ?f)))
              (= (adjoint-pair (opposite ?f)) (LAT.ADJ$opposite (adjoint-pair ?f)))
              (= (instance (opposite ?f)) (type ?f))
              (= (type (opposite ?f)) (instance ?f))))
```

- The function ‘`composition`’ operates on any two concept morphisms that are composable in the sense that the target concept lattice of the first is equal to the source concept lattice of the second. Composition produces a concept morphism, whose components are constructed using composition.

```
(16) (KIF$opspan composable-opspan)
     (= composable-opspan [target source])

(17) (KIF$relation composable)
     (= (KIF$collection1 composable) concept-morphism)
     (= (KIF$collection2 composable) concept-morphism)
     (= (KIF$extent composable) (KIF$pullback composable-opspan))

(18) (KIF$function composition)
     (= (KIF$source composition) (KIF$pullback composable-opspan))
     (= (KIF$target composition) concept-morphism)
     (forall (?f1 (concept-morphism ?f1)
              ?f2 (concept-morphism ?f2) (composable ?f1 ?f2))
         (and (= (source (composition [?f1 ?f2])) (source ?f1))
              (= (target (composition [?f1 ?f2])) (target ?f2))
              (= (adjoint-pair (composition [?f1 ?f2]))
                 (LAT.ADJ$composition [(adjoint-pair ?f2) (adjoint-pair ?f1)]))
              (= (instance (composition [?f1 ?f2]))
                 (SET.FTN$composition [(instance ?f2) (instance ?f1)]))
              (= (type (composition [?f1 ?f2]))
                 (SET.FTN$composition [(type ?f1) (type ?f2)]))))
```

- The function ‘`identity`’ associates a well-defined (identity) concept morphism with any concept lattice – its components are identities.

```
(19) (KIF$function identity)
     (= (KIF$source identity) CL$concept-lattice)
     (= (KIF$target identity) concept-morphism)
     (forall (?l (CL$concept-lattice ?l))
         (and (= (source (identity ?l)) ?l)
              (= (target (identity ?l)) ?l)
              (= (adjoint-pair (identity ?l))
                 (LAT.ADJ$identity (complete-lattice ?l)))
              (= (instance (identity ?l)) (SET.FTN$identity (CL$instance ?l)))
              (= (type (identity ?c)) (SET.FTN$identity (CL$type ?l)))))
```

- A very useful generic concept morphism represents the instance power concept morphism construction. For any class function $f : B \rightarrow A$ the components of the *instance power concept morphism*

  $$\wp f = \langle\langle \wp f, f^{-1}\rangle, f, f^{-1}\rangle : \wp A \rightleftarrows \wp B$$

  over $f$ are defined as follows: the source concept lattice is the instance power concept lattice $\wp A = \langle\langle \wp A, \subseteq_A, \cap_A, \cup_A\rangle, A, \wp A, \{\text{-}\}_A, id_{\wp A}\rangle$ over $A$, the target concept lattice is the instance power concept lattice $\wp B = \langle\langle \wp B, \subseteq_B, \cap_B, \cup_B\rangle, B, \wp B, \{\text{-}\}_B, id_{\wp B}\rangle$ over $B$, the adjoint pair is the (complete lattice) power adjoint pair over $f$, the instance function is $f$, and the type function is the inverse image function $f^{-1} : \wp A \rightarrow \wp B$ from the power-class of $A$ to the power-class of $B$. Note the contravariance.

```
(20) (KIF$function instance-power)
     (= (KIF$source instance-power) SET.FTN$function)
     (= (KIF$target instance-power) concept-morphism)
     (forall (?f (SET.FTN$function ?f))
         (and (= (source (instance-power ?f)) (CL$instance-power (SET.FTN$target ?f)))
              (= (target (instance-power ?f)) (CL$instance-power (SET.FTN$source ?f)))
              (= (adjoint-pair (instance-power ?f)) (LAT.ADJ$power ?f))
              (= (instance (instance-power ?f)) ?f)
              (= (type (instance-power ?f)) (SET.FTN$inverse-image ?f))))
```

## The Namespace of Large Complete Lattices

This namespace will represent large complete lattices, their adjoint pairs and their complete homomorphisms. The terms introduced in this namespace are listed in Table 1.

**Table 1: Terms introduced into the large complete lattice namespace**

| | Collection | Function | Other |
|---|---|---|---|
| `LAT` | `complete-lattice` | `partial-order = underlying`<br>`class meet join`<br>`classification`<br>`cut`<br>`opposite power concept-lattice` | |
| `LAT .ADJ` | `adjoint-pair` | `source target underlying left right`<br>`infomorphism bond`<br>`cut-forward cut-reverse`<br>`opposite composition identity`<br>`power concept-morphism` | `composable-opspan`<br>`composable` |
| `LAT .MOR` | `homomorphism` | `source target function forward reverse`<br>`bonding-pair cut`<br>`opposite composition identity` | `composable-opspan`<br>`composable` |

### *Complete Lattices*

**LAT**

○ A partial order $\boldsymbol{L} = \langle L, \leq_L, \sqcap_L, \sqcup_L \rangle$ is a *complete lattice* when the meet and join exist for all classes $Q \subseteq L$. The underlying partial order is represented by $|\boldsymbol{L}| = \langle L, \leq_L \rangle$. We define a convenience term –the *class* of elements in the complete lattice. These facilitate the expression of some of the axioms below.

```
(1) (KIF$collection complete-lattice)

(2) (KIF$function partial-order)
    (KIF$function underlying)
    (= underlying partial-order)
    (= (KIF$source partial-order) complete-lattice)
    (= (KIF$target partial-order) ORD$partial-order)

(3) (KIF$function class)
    (= (KIF$source class) complete-lattice)
    (= (KIF$target class) SET$class)
    (forall (?l (complete-lattice ?l))
        (= (class ?l) (ORD$class (partial-order ?l))))

(4) (KIF$function meet)
    (= (KIF$source meet) complete-lattice)
    (= (KIF$target meet) SET.FTN$function)
    (forall (?l (complete-lattice ?l))
        (and (= (SET.FTN$source (meet ?l)) (SET$power (class ?l)))
             (= (SET.FTN$target (meet ?l)) (class ?l))
             (forall (?q ((SET$power (class ?l)) ?q))
                 (((greatest (partial-order ?l))
                  ((lower-bound (partial-order ?l)) ?q))
                   ((meet ?l) ?q)))))

(5) (KIF$function join)
    (= (KIF$source join) complete-lattice)
    (= (KIF$target join) SET.FTN$function)
    (forall (?l (complete-lattice ?l))
        (and (= (SET.FTN$source (join ?l)) (SET$power (class ?l)))
             (= (SET.FTN$target (join ?l)) (class ?l))
             (forall (?q ((SET$power (class ?l)) ?q))
                 (((least (partial-order ?l))
                  ((upper-bound (partial-order ?l)) ?q))
                   ((join ?l) ?q)))))
```

- Associated with any complete lattice $\boldsymbol{L} = \langle L, \leq_L, \sqcap_L, \sqcup_L \rangle$ is the classification $\mathsf{cls}(\boldsymbol{L}) = \mathsf{cls}(|\boldsymbol{L}|) = \langle L, L, \leq_L \rangle$, which has $L$-elements as its instances and types, and the lattice order as its incidence.

> The classification associated with a complete lattice is
> – the image of the object function of the bond functor applied to the complete lattice:
> $B$ : COMPLETE ADJOINT $\rightarrow$ BOND.
> – the image of the object function of the bonding pair functor applied to the complete lattice:
> $B^2$ : COMPLETE LATTICE $\rightarrow$ BONDING PAIR.

```
(6) (KIF$function classification)
    (= (KIF$source classification) complete-lattice)
    (= (KIF$target classification) CLS$classification)
    (forall (?l (complete-lattice ?l))
        (= (classification ?l)
           (ORD$classification (partial-order ?l))))
```

- For any complete lattice $\boldsymbol{L} = \langle L, \leq_L, \sqcap_L, \sqcup_L \rangle$ and for every element $l \in L$, the pair $\updownarrow_L(l) = (\downarrow_L l, \uparrow_L l)$ is a formal concept in the concept lattice of the classification of $\boldsymbol{L}$. Hence, there is a special *cut* function $\mathsf{cut}(\boldsymbol{L}) = \updownarrow_L : L \rightarrow \mathsf{lat}(\mathsf{cls}(\boldsymbol{L}))$.

```
(7) (KIF$function cut)
    (= (KIF$source cut) complete-lattice)
    (= (KIF$target cut) SET.FTN$function)
    (forall (?l (complete-lattice ?l))
        (and (SET.FTN$source (cut ?l)) (class ?l))
             (SET.FTN$target (cut ?l)) (CLS.CL$concept (classification ?l)))
             (= (SET.FTN$composition [(cut ?l) (CLS.CL$extent (classification ?l))])
                (ORD$down-embedding (partial-order ?l)))
             (= (SET.FTN$composition [(cut ?l) (CLS.CL$intent (classification ?l))])
                (ORD$up-embedding (partial-order ?l)))))
```

- Here are some preliminary observations that pertain to this cut function.

  On the underlying partial-order of a complete lattice, the composition of the down-embedding and join is the identity on the underlying class. Dually, the composition of the up-embedding and meet is the identity on the underlying class.

```
(forall (?l (complete-lattice ?l))
     (and (= (SET.FTN$composition [(ORD$down-embedding (partial-order ?l)) (join ?l)])
             (SET.FTN$identity (class ?l)))
          (= (SET.FTN$composition [(ORD$up-embedding (partial-order ?l)) (meet ?l)])
             (SET.FTN$identity (class ?l)))))
```

  On the classification of a complete lattice, the instance generation function factors as the composition of join and the above cut function. Dually, the type generation function is the composition of meet followed by cut.

```
(forall (?l (complete-lattice ?l))
        (and (= (CLS.CL$instance-generation (classification ?l))
                (SET.FTN$composition [(join ?l) (cut ?l)]))
             (= (CLS.CL$type-generation (classification ?l))
                (SET.FTN$composition [(meet ?l) (cut ?l)]))))
```

- On the classification of a complete lattice, the instance embedding and type embedding functions are both equal to the above cut function.

```
(forall (?l (complete-lattice ?l))
    (and (= (CLS.CL$instance-embedding (classification ?l))
            (cut ?l))
         (= (CLS.CL$type-embedding (classification ?l))
            (cut ?l))))
```

- It is important to observe that any concept in $\mathsf{lat}(\mathsf{cls}(\boldsymbol{L}))$ is of the form $(\downarrow_L x, \uparrow_L x)$ for some element $x \in L$. In fact, the cut function is a bijection, and its inverse function has two expressions – it is the composition of conceptual extent and join, and it is the composition of conceptual intent and meet.

```
(forall (?l (complete-lattice ?l))
    (and (= (SET.FTN$composition
                [(cut ?l)
                 (SET.FTN$composition
                     [(CLS.CL$extent (classification ?l)) (join ?l)])])
            (SET.FTN$identity (class ?l)))
         (= (SET.FTN$composition
                [(SET.FTN$composition
                     [(CLS.CL$extent (classification ?l)) (join ?l)])
                 (cut ?l)])
            (SET.FTN$identity (CLS.CL$concept (classification ?l))))
         (= (SET.FTN$composition
                [(cut ?l)
                 (SET.FTN$composition
                     [(CLS.CL$intent (classification ?l)) (meet ?l)])])
            (SET.FTN$identity (class ?l)))
         (= (SET.FTN$composition
                [(SET.FTN$composition
                     [(CLS.CL$intent (classification ?l)) (meet ?l)])
                (cut ?l)])
            (SET.FTN$identity (CLS.CL$concept (classification ?l))))))
```

- For any complete lattice $\boldsymbol{L} = \langle L, \leq_L, \sqcap_L, \sqcup_L \rangle$, the *opposite* or *dual* of $\boldsymbol{L}$ is the complete lattice $\boldsymbol{L}^{\perp} = \langle L, \geq_L, \sqcup_L, \sqcap_L \rangle$, whose underlying partial order is the opposite of the underlying partial order of $\boldsymbol{L}$, whose meet is the join of $\boldsymbol{L}$, and whose join is the meet of $\boldsymbol{L}$.

```
(8) (KIF$function opposite)
    (= (KIF$source opposite) complete-lattice)
    (= (KIF$target opposite) complete-lattice)
    (forall (?l (complete-lattice ?l))
        (and (= (partial-order (opposite ?l)) (ORD$opposite (partial-order ?l)))
             (= (meet (opposite ?l)) (join ?l))
             (= (join (opposite ?l)) (meet ?l))))
```

- For any class $C$, the power complete lattice $\wp(C) = \langle \wp C, \subseteq_C, \cap_C, \cup_C \rangle$ is the power class with subclass ordering, intersection as meet and union as join. There is a KIF *power* function that maps a class to its power lattice.

```
(9) (KIF$function power)
    (= (KIF$source power) SET$class)
    (= (KIF$target power) complete-lattice)
    (forall (?c (SET$class ?c))
        (and (= (partial-order (power ?c)) (ORD$power ?c))
             (= (meet (power ?c)) (SET.FTN$intersection ?c))
             (= (join (power ?c)) (SET.FTN$union ?c))))
```

- Any complete lattice $\boldsymbol{L} = \langle L, \leq_L, \sqcap_L, \sqcup_L \rangle$ has an associated *concept lattice* $\boldsymbol{L} = \langle \boldsymbol{L}, L, L, id_{\boldsymbol{L}}, id_{\boldsymbol{L}} \rangle$, where the instance and type classes are the underlying class of the lattice and the instance and type embeddings are the identity function.

```
(10) (KIF$function concept-lattice)
     (= (KIF$source concept-lattice) complete-lattice)
     (= (KIF$target concept-lattice) CL$concept-lattice)
     (forall (?l (complete-lattice ?l))
         (and (= (CL$complete-lattice (concept-lattice ?l)) ?l)
              (= (CL$instance (concept-lattice ?l)) (class ?l))
              (= (CL$type (concept-lattice ?l)) (class ?l))
              (= (CL$instance-embedding (concept-lattice ?l))
                 (SET.FTN$identity (class ?l)))
              (= (CL$type-embedding (concept-lattice ?l))
                 (SET.FTN$identity (class ?l)))))
```

- An easy check shows that the classification of the concept lattice of a complete lattice $\boldsymbol{L}$ is the same as the classification associated with $\boldsymbol{L}$.

```
(forall (?l (complete-lattice ?l))
    (= (CL$classification (concept-lattice ?l))
       (classification ?l)))
```

## *Complete Adjoint*

**LAT.ADJ**

$$\begin{tikzcd} \boldsymbol{K} \arrow[r, "\psi = right(f)"] \arrow[d, dash, "\leq_K"'] \arrow[dr, dash] & \boldsymbol{L} \arrow[d, dash, "\leq_L"] \\ \boldsymbol{K} & \boldsymbol{L} \arrow[l, "\varphi = left(f)"] \end{tikzcd}$$

**Figure 1: Complete Adjoint as an Infomorphism**

○ Complete lattices are related through *adjoint pairs*. This is a restriction to complete lattices of the adjoint pair notion for preorders. For an adjoint pair $\langle\varphi, \psi\rangle : \boldsymbol{L} \rightleftarrows \boldsymbol{K}$ between complete lattices $\boldsymbol{L} = \langle \boldsymbol{L}, \leq_{\boldsymbol{L}}, \sqcap_{\boldsymbol{L}}, \sqcup_{\boldsymbol{L}}\rangle$, and $\boldsymbol{K} = \langle \boldsymbol{K}, \leq_{\boldsymbol{K}}, \sqcap_{\boldsymbol{K}}, \sqcup_{\boldsymbol{K}}\rangle$, the left adjoint $\varphi : \boldsymbol{L} \rightarrow \boldsymbol{K}$ is join-preserving and the right-adjoint $\psi : \boldsymbol{K} \rightarrow \boldsymbol{L}$ is meet-preserving. The two functions determine each other as follows.

$$\varphi(l) = \sqcap_{\boldsymbol{K}} \{k \in K \mid l \leq_{\boldsymbol{L}} \psi(k)\}$$

$$\psi(k) = \sqcup_{\boldsymbol{L}} \{l \in L \mid k \leq_{\boldsymbol{K}} \varphi(l)\}$$

For example, suppose $\psi : \boldsymbol{K} \rightarrow \boldsymbol{L}$ is a meet-preserving monotonic function, and define the function $\varphi : L \rightarrow K$ as above.

– If $l_1 \leq_{\boldsymbol{L}} l_2$ then $\{k \in K \mid l_1 \leq_{\boldsymbol{L}} \psi(k)\} \supseteq \{k \in K \mid l_2 \leq_{\boldsymbol{L}} \psi(k)\}$. Hence, $\varphi(l_1) \leq_{\boldsymbol{K}} \varphi(l_2)$.

– If $l \leq_{\boldsymbol{L}} \psi(k)$ then $k \in \{\tilde{k} \in K \mid l \leq_{\boldsymbol{L}} \psi(\tilde{k})\}$. Hence, $\varphi(l) \leq_{\boldsymbol{K}} k$.

– $\psi(\varphi(l)) = \psi(\sqcap_{\boldsymbol{K}} \{k \in K \mid l \leq_{\boldsymbol{L}} \psi(k)\}) = \sqcap_{\boldsymbol{L}} \{\psi(k) \in L \mid l \leq_{\boldsymbol{L}} \psi(k)\} \geq_{\boldsymbol{L}} l$.

– If $\varphi(l) \leq_{\boldsymbol{K}} k$ then $\psi(\varphi(l)) \leq_{\boldsymbol{L}} \psi(k)$. Hence, $l \leq_{\boldsymbol{L}} \psi(k)$.

Let COMPLETE ADJOINT denote the quasi-category of complete lattices and adjoint pairs.

```
(1) (KIF$collection adjoint-pair)

(2) (KIF$function source)
    (= (KIF$source source) adjoint-pair)
    (= (KIF$target source) LAT$complete-lattice)

(3) (KIF$function target)
    (= (KIF$source target) adjoint-pair)
    (= (KIF$target target) LAT$complete-lattice)

(4) (KIF$function underlying)
    (= (KIF$source underlying) adjoint-pair)
    (= (KIF$target underlying) ORD.ADJ$adjoint-pair)
    (forall (?a (adjoint-pair ?a))
        (and (= (ORD.ADJ$source (underlying ?a)) (LAT$underlying (source ?a)))
             (= (ORD.ADJ$target (underlying ?a)) (LAT$underlying (target ?a)))))
```

o We define the following two terms for convenience of reference.

```
(5) (KIF$function left)
    (= (KIF$source left) adjoint-pair)
    (= (KIF$target left) ORD.FTN$monotonic-function)
    (forall (?a (adjoint-pair ?a))
        (= (left ?a) (ORD.ADJ$left (underlying ?a))))

(6) (KIF$function right)
    (= (KIF$source right) adjoint-pair)
    (= (KIF$target right) ORD.FTN$monotonic-function)
    (forall (?a (adjoint-pair ?a))
        (= (right ?a) (ORD.ADJ$right (underlying ?a))))
```

o Any adjoint pair has an associated *infomorphism* (Figure 1).

```
(7) (KIF$function infomorphism)
    (= (KIF$source infomorphism) adjoint-pair)
    (= (KIF$target infomorphism) CLS.INFO$infomorphism)
    (forall (?a (adjoint-pair ?a))
        (= (infomorphism ?a)
           (ORD.ADJ$infomorphism (underlying ?a))))
```

- Associated with any adjoint pair $\boldsymbol{f} = \langle \mathsf{left}(\boldsymbol{f}), \mathsf{right}(\boldsymbol{f})\rangle = \langle \varphi, \psi\rangle : \boldsymbol{L} \rightleftarrows \boldsymbol{K}$, from complete lattice $\boldsymbol{L}$ to complete lattice $\boldsymbol{K}$, is the *bond* $\mathsf{bnd}(\boldsymbol{f}) : \mathsf{cls}(\boldsymbol{K}) \rightharpoonup \mathsf{cls}(\boldsymbol{L})$ (whose classification relation is) defined by the adjointness property: $l\,\mathsf{bnd}(\boldsymbol{f})k$ iff $\varphi(l) \leq_{\boldsymbol{K}} k$ iff $l \leq_{\boldsymbol{L}} \psi(k)$ for all elements $l \in L$ and $k \in K$. The closure property of bonds is obvious, since $l\,\mathsf{bnd}(\boldsymbol{f}) = \uparrow_{\boldsymbol{K}} \varphi(l)$ for all elements $l \in L$ and $\mathsf{bnd}(\boldsymbol{f})k = \downarrow_{\boldsymbol{L}} \psi(k)$ for all elements $k \in K$. This can equivalently be defined in terms of either the left or the right monotonic function. Here we use the left monotonic function. In particular, we define the classification of the bond via the right operator that maps the left function to a classification relation in the presence of the underlying partial order of the target complete lattice.

> The bond associated with an adjoint pair is the image of the morphism function of the bond functor applied to the adjoint pair:
>
> $B$ : COMPLETE ADJOINT $\rightarrow$ BOND.

```
(8) (KIF$function bond)
    (= (KIF$source bond) adjoint-pair)
    (= (KIF$target bond) CLS.BND$bond)
    (forall (?a (adjoint-pair ?a))
        (and (= (CLS.BND$source (bond ?a)) (LAT$classification (target ?a)))
             (= (CLS.BND$target (bond ?a)) (LAT$classification (source ?a)))
             (= (CLS.BND$classification (bond ?a))
                (SET.FTN$right
                    [(ORD.FTN$function (left ?a))
                     (LAT$partial-order (target ?a))]))))
```

- This bond is the bond of the underlying adjoint pair. This fact could be used as a definition.

```
    (forall (?a (adjoint-pair ?a))
        (= (bond ?a)
           (ORD.ADJ$bond (underlying ?a))))
```

- The *composition* of two *composable* adjoint pairs $\boldsymbol{F} : \boldsymbol{A} \rightarrow \boldsymbol{B}$ and $\boldsymbol{G} : \boldsymbol{B} \rightarrow \boldsymbol{C}$ is the composition of the underlying adjoint pairs.

```
(9) (KIF$opspan composable-opspan)
    (= composable-opspan [target source])

(10) (KIF$relation composable)
     (= (KIF$collection1 composable) adjoint-pair)
     (= (KIF$collection2 composable) adjoint-pair)
     (= (KIF$extent composable) (KIF$pullback composable-opspan))

(11) (KIF$function composition)
     (= (KIF$source composition) (KIF$pullback composable-opspan))
     (= (KIF$target composition) adjoint-pair)
     (forall (?f1 (adjoint-pair ?f1)
              ?f2 (adjoint-pair ?f2) (composable ?f1 ?f2))
         (and (= (source (composition [?f1 ?f2])) (source ?f1))
              (= (target (composition [?f1 ?f2])) (target ?f2))
              (= (underlying (composition [?f1 ?f2]))
                 (ORD.ADJ$composition [(underlying ?f1) (underlying ?f2)]))))
```

- The identity adjoint pair at a complete lattice $\boldsymbol{L}$ is the identity adjoint pair of the underlying order.

```
(12) (KIF$function identity)
     (= (KIF$source identity) LAT$complete-lattice)
     (= (KIF$target identity) adjoint-pair)
     (forall (?l (LAT$complete-lattice ?l))
         (and (= (source (identity ?l)) ?l)
              (= (target (identity ?l)) ?l)
              (= (underlying (identity ?l))
                 (ORD.ADJ$identity (LAT$underlying ?l)))))
```

- In section 3.2 of the paper (Kent 2001), which is concerned with relational equivalence, the following ideas are introduced and developed in order to demonstrate the categorical equivalence BOND $\equiv$

COMPLETE ADJOINT. The cut monotonic function $\boldsymbol{L} \rightarrow \mathsf{A}(\mathsf{B}(\boldsymbol{L}))$ was defined in the complete lattice namespace.

- The *cut-forward* adjoint pair $\langle \mathit{extent}(\mathit{cls}(\boldsymbol{L})) \cdot \mathit{join}(\boldsymbol{L}), \mathit{cut}(\boldsymbol{L})\rangle : \mathit{lat}(\mathit{cls}(\boldsymbol{L})) \rightleftarrows \boldsymbol{L}$ has the cut function as its right monotonic function and the composition of extent and join (or the composition of intent and meet) as its left monotonic function. This adjoint pair is a pair of inverse functions, and hence is an isomorphism from the complete lattice of its associated classification $\mathsf{A}(\mathsf{B}(\boldsymbol{L}))$ to a complete lattice $\boldsymbol{L}$.
- The *cut-reverse* adjoint pair $\langle \mathit{cut}(\boldsymbol{L}), \mathit{extent}(\mathit{cls}(\boldsymbol{L})) \cdot \mathit{join}(\boldsymbol{L})\rangle : \boldsymbol{L} \rightleftarrows \mathit{lat}(\mathit{cls}(\boldsymbol{L}))$ flips the inverses – it has the cut function as its left monotonic function and the composition of extent and join as its right monotonic function.

This adjoint pair is a pair of inverse functions, and hence is an isomorphism from a complete lattice $\boldsymbol{L}$ to the complete lattice of its associated classification $\mathit{lat}(\mathit{cls}(\boldsymbol{L}))$.

```
(13) (KIF$function cut-forward)
     (= (KIF$source cut-forward) LAT$complete-lattice)
     (= (KIF$target cut-forward) adjoint-pair)
     (forall (?l (LAT$complete-lattice ?l))
         (and (= (source (cut-forward ?l))
                 (CLS.CL$complete-lattice (LAT$classification ?l)))
              (= (target (cut-forward ?l)) ?l)
              (= (ORD.FTN$function (left (cut-forward ?l)))
                 (SET.FTN$composition
                     [(CLS.CL$extent (LAT$classification ?l))
                      (LAT$join ?l)]))
              (= (ORD.FTN$function (right (cut-forward ?l))) (LAT$cut ?l))))

(14) (KIF$function cut-reverse)
     (= (KIF$source cut-reverse) LAT$complete-lattice)
     (= (KIF$target cut-reverse) adjoint-pair)
     (forall (?l (complete-lattice ?l))
         (and (= (source (cut-reverse ?l)) ?l)
              (= (target (cut-reverse ?l))
                 (CLS.CL$complete-lattice (LAT$classification ?l)))
              (= (ORD.FTN$function (left (cut-reverse ?l))) (LAT$cut ?l))
              (= (ORD.FTN$function (right (cut-reverse ?l)))
                 (SET.FTN$composition
                     [(CLS.CL$extent (LAT$classification ?l))
                      (LAT$join ?l)]))))
```

○ Let $\boldsymbol{f} = \langle \varphi, \psi\rangle : \boldsymbol{L} \rightleftarrows \boldsymbol{K}$ be a complete adjoint, an adjoint pair of monotonic functions, between complete lattices $\boldsymbol{L} = \langle L, \leq_L, \wedge_L, \vee_L\rangle$, and $\boldsymbol{K} = \langle K, \leq_K, \wedge_K, \vee_K\rangle$ with associated bond $\mathit{bnd}(\boldsymbol{f}) : \mathit{cls}(\boldsymbol{L}) \rightarrow \mathit{cls}(\boldsymbol{K})$. Then, the right adjoint of $\mathit{adj}(\mathit{bnd}(\boldsymbol{f})$ maps $(\downarrow_L x, \uparrow_L x) \mapsto (\downarrow_K \psi(x), \uparrow_K \psi(x))$ and the left adjoint of $\mathit{adj}(\mathit{bnd}(\boldsymbol{f})$ maps $(\downarrow_K y, \uparrow_K y) \mapsto (\downarrow_L \varphi(y), \uparrow_L \varphi(y))$. This is equivalent to the natural isomorphism (commuting Diagram 1):

$$\mathit{cut\text{-}reverse}(\boldsymbol{L}) \cdot \mathit{adj}(\mathit{bnd}(\boldsymbol{f})) = \boldsymbol{f} \cdot \mathit{cut\text{-}reverse}(\boldsymbol{K}).$$

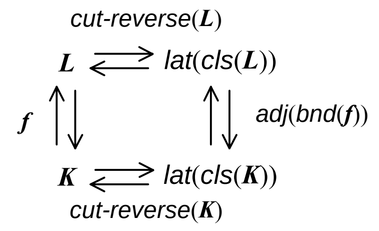

$$\begin{tikzcd}
\boldsymbol{L} \arrow[r, shift left, "\mathit{cut\text{-}reverse}(\boldsymbol{L})"] \arrow[d, shift left] & \mathit{lat}(\mathit{cls}(\boldsymbol{L})) \arrow[l, shift left] \arrow[d, shift left] \\
\boldsymbol{K} \arrow[r, shift left] \arrow[u, shift left, "\boldsymbol{f}"] & \mathit{lat}(\mathit{cls}(\boldsymbol{K})) \arrow[l, shift left, "\mathit{cut\text{-}reverse}(\boldsymbol{K})"] \arrow[u, shift left, "\mathit{adj}(\mathit{bnd}(\boldsymbol{f}))"']
\end{tikzcd}$$

**Diagram 1: Natural Isomorphism**

So, up to isomorphism, $\mathit{adj}(\mathit{bnd}(\boldsymbol{f})$ is the same as $\boldsymbol{f}$.

For any complete lattice $\boldsymbol{L}$ the cut-reverse adjoint pair $\mathit{cut\text{-}reverse}(\boldsymbol{L})$ is the $\boldsymbol{L}^{\text{th}}$ component of a natural isomorphism $\boldsymbol{L} \cong \mathsf{A}(\mathsf{B}(\boldsymbol{L}))$, demonstrating that the quasi-category of complete adjoints is categorically equivalent to the quasi-category of bonds:

COMPLETE ADJOINT ≡ BOND.

```
(forall (?a (adjoint-pair ?a))
    (= (composition (cut-reverse (source ?a)) (adjoint-pair (bond ?a)))
       (composition ?a (cut-reverse (target ?a)))))
```

- Duality can be extended to adjoint pairs. For any adjoint pair $\boldsymbol{f} : \boldsymbol{L} \rightleftarrows \boldsymbol{K}$, the *opposite* or *dual* of $\boldsymbol{f}$ is the adjoint pair $\boldsymbol{f}^{\perp} : \boldsymbol{K}^{\perp} \rightleftarrows \boldsymbol{L}^{\perp}$, whose source complete lattice is the opposite of the target of $\boldsymbol{f}$, whose target complete lattice is the opposite of the source of $\boldsymbol{f}$, whose underlying adjoint pair is the opposite of the adjoint pair of $\boldsymbol{f}$.

```
(15) (KIF$function opposite)
     (= (KIF$source opposite) adjoint-pair)
     (= (KIF$target opposite) adjoint-pair)
     (forall (?f (adjoint-pair ?f))
         (and (= (source (opposite ?f)) (CL$opposite (target ?f)))
              (= (target (opposite ?f)) (CL$opposite (source ?f)))
              (= (underlying (opposite ?f)) (ORD.ADJ$opposite (underlying ?f)))))
```

- For any class function $f : A \rightarrow B$ there is a *power adjoint pair* $\wp f : \wp A \rightleftarrows \wp B$ over $f$ defined as follows: the source is the complete lattice $\wp(A) = \langle \wp A, \subseteq_A, \cap_A, \cup_A \rangle$, the target is the complete lattice $\wp(B) = \langle \wp B, \subseteq_B, \cap_B, \cup_B \rangle$, the left monotonic function is the direct image function $\wp f : \wp A \rightarrow \wp B$, whose right monotonic function is the inverse image function $f^{-1} : \wp B \rightarrow \wp A$, and whose fundamental property holds, since the following equivalence holds

  $\wp f(X) \subseteq_B Y$ iff $X \subseteq_A f^{-1}(Y)$

  for all subclasses $X \subseteq A$ and $Y \subseteq B$.

```
(16) (KIF$function power)
     (= (KIF$source power) SET.FTN$function)
     (= (KIF$target power) adjoint-pair)
      (forall (?f (SET.FTN$function ?f))
          (and (= (source (power ?f)) (LAT$power (SET.FTN$source ?f)))
               (= (target (power ?f)) (LAT$power (SET.FTN$target ?f)))
               (= (left (power ?f)) (SET.FTN$direct-image ?f))
               (= (right (power ?f)) (SET.FTN$inverse-image ?f))))
```

- For any adjoint pair $\boldsymbol{f} : \boldsymbol{L} = \langle L, \leq_L, \sqcap_L, \sqcup_L \rangle \rightleftarrows \boldsymbol{K} = \langle K, \leq_K, \sqcap_K, \sqcup_K \rangle$ between complete lattices there is a associated concept morphism $\boldsymbol{f} : \langle \boldsymbol{K}, K, K, id_{\boldsymbol{K}}, id_{\boldsymbol{K}} \rangle \rightleftarrows \langle \boldsymbol{L}, L, L, id_{\boldsymbol{L}}, id_{\boldsymbol{L}} \rangle$ (in the reverse direction) between the concept lattices associated with the target and source complete lattices.

```
(17) (KIF$function concept-morphism)
     (= (KIF$source concept-morphism) adjoint-pair)
     (= (KIF$target concept-morphism) CL.MOR$concept-morphism)
     (forall (?f (adjoint-pair ?f))
         (and (= (CL.MOR$source (concept-morphism ?f))
                 (LAT$concept-lattice (target ?f)))
              (= (CL.MOR$target (concept-morphism ?f))
                 (LAT$concept-lattice (source ?f)))
              (= (CL.MOR$adjoint-pair (concept-morphism ?f)) ?f)
              (= (CL.MOR$instance (concept-morphism ?f))
                 (ORD.ADJ$left (underlying ?f)))
              (= (CL.MOR$type (concept-morphism ?f))
                 (ORD.ADJ$right (underlying ?f)))))
```

- An easy check shows that the infomorphism of the concept morphism of an adjoint pair $\boldsymbol{f}$ is the same as the infomorphism associated with $\boldsymbol{f}$.

```
(forall (?f (adjoint-pair ?f))
    (= (CL.MOR$infomorphism (concept-morphism ?f))
       (infomorphism ?f)))
```

## *Complete Lattice Homomorphism*

**LAT.MOR**

$$\begin{tikzcd} K \arrow[r, "\theta"] \arrow[d, dash, "\leq_K"'] \arrow[dr, phantom, "\textit{reverse}"] & L \arrow[d, dash, "\leq_L"] \\ K & L \arrow[l, "\psi"] \end{tikzcd}$$

$$\begin{tikzcd} L \arrow[r, "\psi"] \arrow[d, dash, "\leq_L"'] \arrow[dr, phantom, "\textit{forward}"] & K \arrow[d, dash, "\leq_K"] \\ L & K \arrow[l, "\varphi"] \end{tikzcd}$$

**Figure 2: Complete Homomorphism**

Unfortunately, adjoint pairs are not the best morphisms for making structural comparisons between complete lattices. Another morphism between complete lattices called complete homomorphisms are best for this.

○ A *homomorphism* $\psi : \boldsymbol{L} \rightarrow \boldsymbol{K}$ between complete lattices $\boldsymbol{L}$ and $\boldsymbol{K}$ is a monotonic function that preserves both joins and meets (Figure 2). Being meet-preserving, $\psi$ has a left adjoint $\varphi : K \rightarrow L$, and being join-preserving $\psi$ has a right adjoint $\theta : K \rightarrow L$. Therefore, a complete homomorphism is the middle monotonic function in two adjunctions $\varphi \dashv \psi \dashv \theta$. Since it is more algebraic, we use the latter adjoint pair characterization in the definition of a complete lattice homomorphism. Let COMPLETE LATTICE denote the quasi-category of complete lattices and complete homomorphisms.

```
(1) (KIF$collection homomorphism)

(2) (KIF$function source)
    (= (KIF$source source) homomorphism)
    (= (KIF$target source) LAT$complete-lattice)

(3) (KIF$function target)
    (= (KIF$source target) homomorphism)
    (= (KIF$target target) LAT$complete-lattice)

(4) (KIF$function function)
    (= (KIF$source function) homomorphism)
    (= (KIF$target function) ORD.FTN$monotonic-function)

(5) (KIF$function forward)
    (= (KIF$source forward) homomorphism)
    (= (KIF$target forward) LAT.ADJ$adjoint-pair)
    (forall (?h (homomorphism ?h))
        (and (= (LAT.ADJ$source (forward ?h)) (target ?h))
             (= (LAT.ADJ$target (forward ?h)) (source ?h))
             (= (LAT.ADJ$right (forward ?h)) (function ?h))))

(6) (KIF$function reverse)
    (= (KIF$source reverse) homomorphism)
    (= (KIF$target reverse) LAT.ADJ$adjoint-pair)
    (forall (?h (homomorphism ?h))
        (and (= (LAT.ADJ$source (reverse ?h)) (source ?h))
             (= (LAT.ADJ$target (reverse ?h)) (target ?h))
             (= (LAT.ADJ$left (reverse ?h)) (function ?h))))
```

○ For each complete lattice homomorphism $\psi : \boldsymbol{L} \rightarrow \boldsymbol{K}$ there is an associated *bonding pair*.

The bonding pair associated with a complete lattice homomorphism is the image of the morphism function of the bonding pair functor applied to the complete lattice homomorphism:

$B^2$ : COMPLETE LATTICE $\rightarrow$ BONDING PAIR.

```
(7) (KIF$function bonding-pair)
    (= (KIF$source bonding-pair) homomorphism)
    (= (KIF$target bonding-pair) CLS.BNDPR$bonding-pair)
    (forall (?h (homomorphism ?h))
        (and (= (CLS.BND$source (bonding-pair ?h)) (LAT$classification (source ?a)))
             (= (CLS.BND$target (bonding-pair ?h)) (LAT$classification (target ?a)))
             (= (CLS.BND$forward (bonding-pair ?h)) (LAT.ADJ$bond (forward ?h)))
             (= (CLS.BND$reverse (bonding-pair ?h)) (LAT.ADJ$bond (reverse ?h)))))
```

- The *composition* of two *composable* homomorphisms $\psi_1 : \boldsymbol{L} \rightarrow \boldsymbol{K}$ and $\psi_2 : \boldsymbol{K} \rightarrow \boldsymbol{M}$ is the composition of the underlying function and forward and reverse adjoint pairs.

```
(8) (KIF$opspan composable-opspan)
    (= composable-opspan [target source])

(9) (KIF$relation composable)
    (= (KIF$collection1 composable) homomorphism)
    (= (KIF$collection2 composable) homomorphism)
    (= (KIF$extent composable) (KIF$pullback composable-opspan))

(10) (KIF$function composition)
     (= (KIF$source composition) (KIF$pullback composable-opspan))
     (= (KIF$target composition) adjoint-pair)
     (forall (?h1 (homomorphism ?h1)
              ?h2 (homomorphism ?h2) (composable ?h1 ?h2))
         (and (= (source (composition [?h1 ?h2])) (source ?h1))
              (= (target (composition [?h1 ?h2])) (target ?h2))
              (= (function (composition [?h1 ?h2]))
                 (ORD.FTN$composition [(function ?h1) (function ?h2)]))
              (= (forward (composition [?h1 ?h2]))
                 (LAT.ADJ$composition [(forward ?h2) (forward ?h1)]))
              (= (reverse (composition [?h1 ?h2]))
                 (LAT.ADJ$composition [(reverse ?h1) (reverse ?h2)]))))
```

- The *identity* homomorphism at a complete lattice $\boldsymbol{L}$ is the identity monotonic function of the underlying order.

```
(11) (KIF$function identity)
     (= (KIF$source identity) LAT$complete-lattice)
     (= (KIF$target identity) homomorphism)
     (forall (?l (LAT$complete-lattice ?l))
         (and (= (source (identity ?l)) ?l)
              (= (target (identity ?l)) ?l)
              (= (function (identity ?l)) (ORD.FTN$identity (LAT$underlying ?l)))
              (= (forward (identity ?l)) (LAT.ADJ$identity ?l))
              (= (reverese (identity ?l)) (LAT.ADJ$identity ?l))))
```

- Duality can be extended to complete homomorphisms. For any homomorphism $\psi : \boldsymbol{L} \rightarrow \boldsymbol{K}$, the *opposite* or *dual* of $\psi$ is the complete homomorphism $\psi^{\perp} : \boldsymbol{K}^{\perp} \rightleftarrows \boldsymbol{L}^{\perp}$, whose source complete lattice is the opposite of the target of $\boldsymbol{f}$, whose target complete lattice is the opposite of the source of $\boldsymbol{f}$, whose forward adjoint pair is the opposite of the reverse adjoint pair of $\boldsymbol{f}$, and whose reverse adjoint pair is the opposite of the forward adjoint pair of $\boldsymbol{f}$.

```
(12) (KIF$function opposite)
     (= (KIF$source opposite) adjoint-pair)
     (= (KIF$target opposite) adjoint-pair)
     (forall (?f (adjoint-pair ?f))
         (and (= (source (opposite ?f)) (CL$opposite (target ?f)))
              (= (target (opposite ?f)) (CL$opposite (source ?f)))
              (= (function (opposite ?f)) (ORD.FTN$opposite (function ?f)))
              (= (forward (opposite ?f)) (ORD.ADJ$opposite (reverse ?f)))
              (= (reverse (opposite ?f)) (ORD.ADJ$opposite (forward ?f)))))
```

- In section 3.3 of the paper (Kent 2001), which is concerned with complete relational equivalence, the following ideas are introduced and developed in order to demonstrate the categorical equivalence BONDING PAIR ≡ COMPLETE LATTICE. The *cut* complete lattice homomorphism $\psi : \boldsymbol{L} \rightarrow lat(cls(\boldsymbol{L}))$ is a bijection from a complete lattice $\boldsymbol{L}$ to the complete lattice of its classification $lat(cls(\boldsymbol{L}))$. Its forward adjoint pair is the cut-forward adjoint pair and its reverse adjoint pair is the cut-reverse adjoint pair.

$$\begin{tikzcd}
\boldsymbol{L} \arrow[r, "cut(\boldsymbol{L})"] \arrow[d, "\psi"'] & lat(cls(\boldsymbol{L})) \arrow[d, "adj(bndpr(\psi))"] \\
\boldsymbol{K} \arrow[r, "cut(\boldsymbol{K})"'] & lat(cls(\boldsymbol{K}))
\end{tikzcd}$$

**Diagram 2: Natural Isomorphism**

```
(13) (KIF$function cut)
     (= (KIF$source cut) LAT$complete-lattice)
     (= (KIF$target cut) homomorphism)
```

```
(forall (?l (complete-lattice ?l))
    (and (= (source (cut ?l)) ?l)
         (= (target (cut ?l)) (CLS.CL$complete-lattice (LAT$classification ?l)))
         (= (forward (cut ?l)) (LAT.ADJ$cut-forward ?l))
         (= (reverse (cut ?l)) (LAT.ADJ$cut-reverse ?l))))
```

○ Now consider any complete lattice homomorphism $\psi : \boldsymbol{L} \rightarrow \boldsymbol{K}$ between complete lattices $\boldsymbol{L}$ and $\boldsymbol{K}$ with associated adjunctions $\varphi \dashv \psi \dashv \theta$. The bonding pair functor maps this to the bonding pair *bndpr*($\psi$) = ($\mathbf{B}(\langle\varphi, \psi\rangle)$, $\mathbf{B}(\langle\psi, \theta\rangle)$), and the complete lattice functor maps this to the complete homomorphism *adj*(*bndpr*($\psi$)) = $\tilde{\psi} : \mathbf{L}(\langle \boldsymbol{L}, \boldsymbol{L}, \leq_{\boldsymbol{L}}\rangle) \rightarrow \mathbf{L}(\langle \boldsymbol{K}, \boldsymbol{K}, \leq_{\boldsymbol{K}}\rangle)$ with associated adjunctions $\tilde{\varphi} \dashv \tilde{\psi} \dashv \tilde{\theta}$, where $\tilde{\varphi}((\downarrow_{\boldsymbol{K}} y, \uparrow_{\boldsymbol{K}} y)) = (\downarrow_{\boldsymbol{L}} \varphi(y), \uparrow_{\boldsymbol{L}} \varphi(y))$, $\tilde{\psi}((\downarrow_{\boldsymbol{L}} x, \uparrow_{\boldsymbol{L}} x)) = (\downarrow_{\boldsymbol{K}} \psi(x), \uparrow_{\boldsymbol{K}} \psi(x))$ and $\tilde{\theta}((\downarrow_{\boldsymbol{K}} y, \uparrow_{\boldsymbol{K}} y)) = (\downarrow_{\boldsymbol{L}} \theta(y), \uparrow_{\boldsymbol{L}} \theta(y))$. Clearly, the naturality condition holds (commuting Diagram 2) between $\psi$ and *adj*(*bndpr*($\psi$)).

For any complete lattice $\boldsymbol{L}$ the cut complete lattice homomorphism *cut*($\boldsymbol{L}$) is the $\boldsymbol{L}$th component of a natural isomorphism $\boldsymbol{L} \cong A^2(B^2(\boldsymbol{L}))$, demonstrating that the quasi-category of complete lattices is categorically equivalent to the quasi-category of bonding pairs:

COMPLETE LATTICE ≡ BONDING PAIR.

```
(forall (?h (homomorphism ?h))
   (= (composition [(cut (source ?h)) (CLS.BNDPR$homomorphism (bonding-pair ?h))])
      (composition [?h (cut (target ?h))])))
```

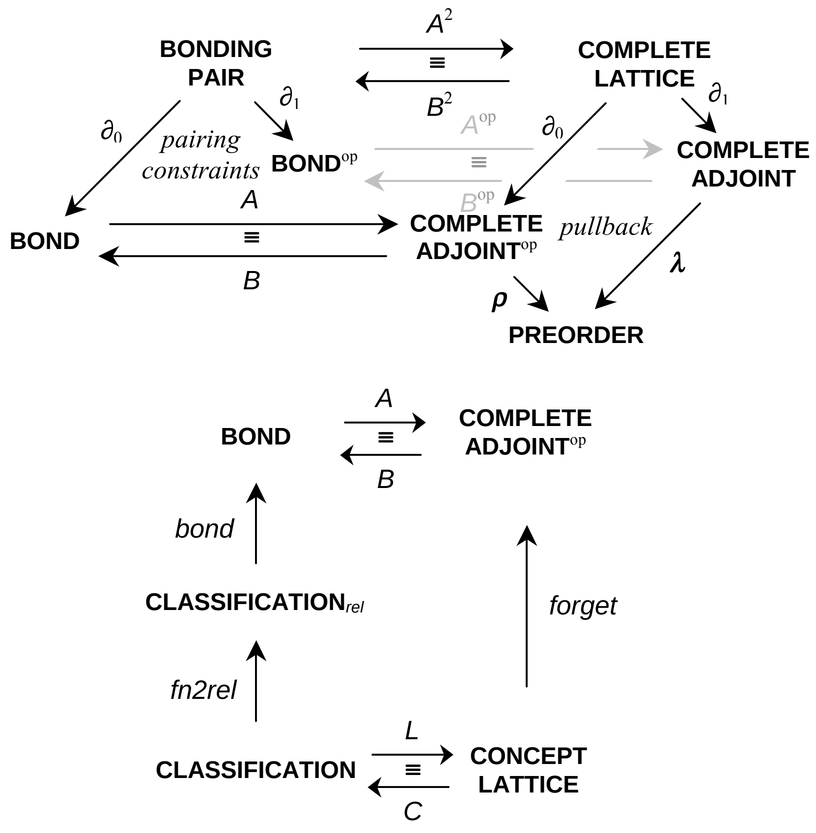


**Figure 1: Architectural Diagram of Distributed Conceptual Structures (functors and natural isomorphisms)**

**Table 1: Mathematical–Ontological Correspondences in the Architectural Diagram**

| **Mathematical Notation** | | **Ontological Terminology** |
|---|---|---|
| $A^2$<br>*Complete Lattice Functor* | object part:<br>morphism part: | ‘`CLS.CL$complete-lattice`’<br>‘`CLS.BNDPR$homomorphism`’ |
| $B^2$<br>*Bonding Pair Functor* | object part:<br>morphism part: | ‘`LAT$classification`’<br>‘`LAT.MOR$bonding-pair`’ |
| $Id_{\text{COMPLETE LATTICE}} \equiv B^2 \circ A^2$<br>*Cut Natural Isomorphism* | component: | ‘`LAT.MOR$cut`’ |
| $A^2 \circ B^2 \equiv Id_{\text{BONDING PAIR}}$<br>*Iota-Tau Natural Isomorphism* | component: | ‘`CLS.BNDPR$iota-tau`’ |
| $A$<br>*Complete Adjoint Functor* | object part:<br>morphism part: | ‘`CLS.CL$complete-lattice`’<br>‘`CLS.BND$bond`’ |
| $B$<br>*Bond Functor* | object part:<br>morphism part: | ‘`LAT$classification`’<br>‘`LAT.ADJ$bond`’ |
| $Id_{\text{COMPLETE ADJOINT}} \equiv B \circ A$<br>*Cut-Reverse Natural Isomorphism* | component: | ‘`LAT.ADJ$cut-reverse`’ |
| $A \circ B \equiv Id_{\text{BOND}}$<br>*Iota Natural Isomorphism* | component: | ‘`CLS.BND$iota`’ (and ‘`CLS.BND$tau`’) |
| $L$<br>*Concept Lattice Functor* | object part:<br>morphism part: | ‘`CLS.CL$concept-lattice`’<br>‘`CLS.INFO$concept-morphism`’ |
| $C$<br>*Classification Functor* | object part:<br>morphism part: | ‘`CL$classification`’<br>‘`CL.MOR$infomorphism`’ |
| $Id_{\text{CONCEPT LATTICE}} \equiv C \circ L$<br>*Representation Natural Isomorphism* | component: | ‘`CL.MOR$representation`’ |
| $L \circ C = Id_{\text{CLASSIFICATION}}$<br>*equality* | | |
| $\partial_0$<br>*0th Bond Projection Functor* | object part:<br>morphism part: | implicit identity function<br>‘`CLS.BNDPR$forward`’ |
| $\partial_1$<br>*1st Bond Projection Functor* | object part:<br>morphism part: | implicit identity function<br>‘`CLS.BNDPR$reverse`’ |
| $\partial_0$<br>*0th Adjoint Pair Projection Functor* | object part:<br>morphism part: | implicit identity function<br>‘`LAT.MOR$forward`’ |
| $\partial_1$<br>*1st Adjoint Pair Projection Functor* | object part:<br>morphism part: | implicit identity function<br>‘`LAT.MOR$reverse`’ |
| $\lambda$<br>*Left Projection Functor* | object part:<br>morphism part: | ‘`LAT$underlying`’<br>‘`LAT.ADJ$left`’ |
| $\rho$<br>*Right Projection Functor* | object part:<br>morphism part: | ‘`LAT$underlying`’<br>‘`LAT.ADJ$right`’ |

- The *complete lattice functor* $A^2$ : BONDING PAIR → COMPLETE LATTICE is the operator that maps a classification $\boldsymbol{A}$ to its concept lattice $A^2(\boldsymbol{A}) \triangleq L(\boldsymbol{A})$ regarded as a complete lattice only, and maps a bonding pair $\langle \boldsymbol{F}, \boldsymbol{G} \rangle : \boldsymbol{A} \rightleftharpoons \boldsymbol{B}$ to its complete lattice homomorphism $A^2(\langle \boldsymbol{F}, \boldsymbol{G} \rangle) \triangleq \psi_F = \varphi_G : L(\boldsymbol{A}) \rightarrow L(\boldsymbol{B})$.
- The *bonding pair functor* $B^2$ : COMPLETE LATTICE → BONDING PAIR is the operator that maps a complete lattice $\boldsymbol{L}$ to its classification $\langle L, L, \leq_{\boldsymbol{L}} \rangle$ and maps a complete lattice homomorphism to its bonding pair as above. Since the bond functor $B$ is functorial, so is $B^2$.

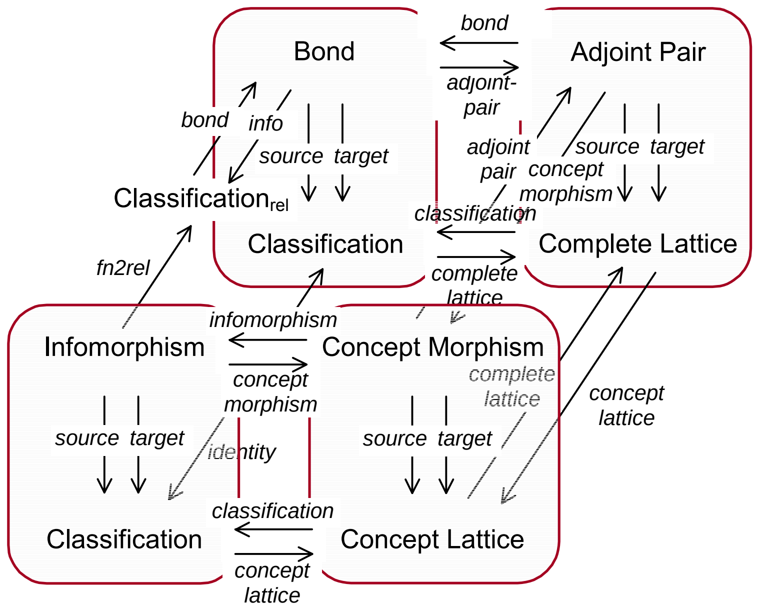


**Diagram 1: Core Collections and Functions in the Architectural Diagram**

## The Namespace of Large Classifications

This is the namespace for large classification and their morphisms: functional/relational infomorphisms, bonds and bonding pairs. In addition to strict classification terminology and axioms, this namespace will provide a bridge from classifications and their morphisms to complete/concept lattices and their morphisms. The terms introduced in this namespace are listed in Table 1.

**Table 1: Terms introduced into the large classification namespace**

| | Collection | Function | Other |
|---|---|---|---|
| `CLS` | `classification` `separated` `extensional` | `instance` `type` `incidence` `extent` `intent` `indistinguishable` `coextensive` `opposite` `instance-power` | |
| `CLS .CL` | | `left-derivation` `right-derivation` `instance-closure` `type-closure` `concept` `extent` `intent` `instance-generation` `type-generation` `concept-order` `meet` `join` `complete-lattice` `coreflection` `reflection` `instance-embedding` `type-embedding` `instance-concept` `type-concept` `instance-order` `type-order` `concept-lattice` | `truth-classification` `truth-concept-lattice` `type-closed` `instance-closed` |
| `CLS .FIB` | | `instance` `instance-index` `type` `type-index` `left-derivation` `right-derivation` `instance-closure` `type-closure` `concept` `extent` `intent` `index` `instance-generation` `type-generation` | |
| `CLS .COLL` | `concept` | `classification` `index` `extent` `intent` `opposite` `2-cell` `source` `target` `index-function` `terminal` `mediator-function` `mediator` `inverse-image` `inverse-image-mediator` `instance-distribution` `type-distribution` | `inverse-image-opspan` `invertible` |
| `CLS .INFO` | `infomorphism` | `source` `target` `instance` `type` `bond` `monotonic-instance` `monotonic-type` `relational-instance` `relational-type` `relational-infomorphism` `fn2rel` `opposite` `composition` `identity` `instance-power` `eta` `adjoint-pair` `concept-morphism` | `composable-image-opspan` `composable` |
| `CLS .REL` | `infomorphism` | `source` `target` `instance` `type` `bond` `opposite` `composition` `identity` | `composable-image-opspan` `composable` |
| `CLS .BND` | `bond` | `source` `target` `classification` `bimodule` `infomorphism` `adjoint-pair` `opposite` `composition` `identity` `iota` `tau` | `composable-image-opspan` `composable` |
| `CLS .BNDPR` | `bonding-pair` | `source` `target` `forward` `reverse` `conceptual-image` `opposite` `composition` `identity` `tau-iota` `iota-tau` `homomorphism` | `composable-image-opspan` `composable` |
| `CLS .COL` | | `counique` | `initial` |

| | | | |
|---|---|---|---|
| CLS<br>.COL<br>.COPRD | diagram<br>pair<br>cocone | classification1 classification2<br>instance-diagram instance-pair<br>type-diagram type-pair<br>opposite<br>cocone-diagram opvertex opfirst opsecond<br>instance-cone type-cocone<br>colimiting-cocone colimit binary-coproduct<br>injection1 injection2<br>comediator | |
| CLS<br>.COL<br>.COINV | coinvariant | classification base class endorelation<br>coquotient canon comediator | respects |
| CLS<br>.COL<br>.COEQ | diagram<br>parallel-pair<br>cocone | source target<br>infomorphism1 infomorphism2<br>instance-diagram instance-parallel-pair<br>type-diagram type-parallel-pair<br>coinvariant<br>cocone-diagram opvertex infomorphism<br>instance-cone type-cocone<br>colimiting-cocone colimit coequalizer canon | |
| CLS<br>.COL<br>.PSH | diagram<br>span<br>cocone | classification1 classification2 vertex<br>first second<br>pair opposite<br>instance-diagram instance-opspan<br>type-diagram type-span<br>coequalizer-diagram parallel-pair<br>cocone-diagram opvertex opfirst opsecond<br>binary-coproduct-cocone<br>coequalizer-cocone<br>colimiting-cocone colimit pushout injec-<br>tion1 injection2 comediator | |

Table 2 lists the correspondence between standard mathematical notation and the ontological terminology in the namespace for classifications and functional/relation infomorphisms and bonds.

**Table 2: Correspondence between Mathematical Notation and Ontological Terminology**

| Mathematical Notation | Ontological Terminology | Natural Language Description |
|---|---|---|
| **`CLS`** | | |
| $A = \langle tok(A), typ(A), \models_A \rangle$ | `‘classification’` | a *classification* – identified with a binary relation; it is determined by its three components |
| $tok(A)$ | `‘instance’` | the *instance* (token) class of a classification – identified with the source class of a binary relation |
| $typ(A)$ | `‘type’` | the *type* class of a classification – identified with the target class of a binary relation |
| $\models_A$ | `‘incidence’` | the *incidence* (or *classification*) class of a classification – identified with the extent class of a binary relation |
| $\wp$ | `‘instance-power’` | the *instance power* operator on classes – this maps classes to classifications |
| $i_1 \sim_A i_2$ | `‘indistinguishable’` | the information flow *indistinguishable* relation on instances |
| $t_1 \sim_A t_2$ | `‘coextensive’` | the information flow *coextensive* relation on types |
| “separated” | `‘separated’` | the *separated* subcollection of classifications |
| “extensional” | `‘extensional’` | the *extensional* subcollection of classifications |
| $(\text{-})^{\propto}$ or $(\text{-})^{\perp}$ or $(\text{-})^{\text{op}}$ | `‘opposite’` | the *involution* (or *transpose* or *opposite* or *dual*) operator on classifications |
| **`CLS.CL`** | | |
| $(\text{-})'$ or $(\text{-})^{A}$ | `‘left-derivation’`, `‘right-derivation’` | *left/right derivation* operations for a classification |
| $(\text{-})''$ or $(\text{-})^{AA}$ | `‘instance-closure’` `‘type-closure’` | *instance/type closure* operations for a classification |
| $c = (X, Y)$ | `‘concept’` | a *formal concept* for a classification |
| $c_1 \leq_A c_2$ | `‘concept-order’` | the concept order of a classification – this is the partial order underlying the concept lattice of a classification |
| $\underline{\mathfrak{B}}A$ | `‘concept-lattice’` | the *concept lattice* of a classification – the German word for “formal concepts” is *die begriffe* |
| $\gamma_A : tok(B) \rightarrow \underline{\mathfrak{B}}A$ | `‘instance-embedding’` | the *instance embedding function* – maps an instance to the concept that it generates |
| $\mu_A : typ(B) \rightarrow \underline{\mathfrak{B}}A$ | `‘type-embedding’` | the *type embedding function* – maps a type to the concept that it generates |
| **`CLS.INFO`** | | |
| $f = \langle f^{\wedge}, f^{\vee} \rangle : A \rightleftarrows B$ | `‘infomorphism’` | an *infomorphism* from classification $A$ to classification $B$ – determined by its instance and type functions |
| $f^{\wedge} : typ(A) \rightarrow typ(B)$ | `‘type’` | the *type* function of an infomorphism |
| $f^{\vee} : tok(B) \rightarrow tok(A)$ | `‘instance’` | the *instance* (or token) function of an infomorphism |
| $gf : A \rightleftarrows C$ | `‘composition’` | the *composition* of two composable infomorphisms $f : A \rightleftarrows B$ and $g : B \rightleftarrows C$ |
| $1_A : A \rightleftarrows A$ | `‘identity’` | the *identity* infomorphism on classification $A$ |
| $(\text{-})^{\propto}$ or $(\text{-})^{\perp}$ or $(\text{-})^{\text{op}}$ | `‘opposite’` | the *involution* (or *transpose* or *opposite* or *dual*) operator on infomorphisms |
| $\wp$ | `‘instance-power’` | the *instance power* operator on functions – this maps functions to infomorphisms |

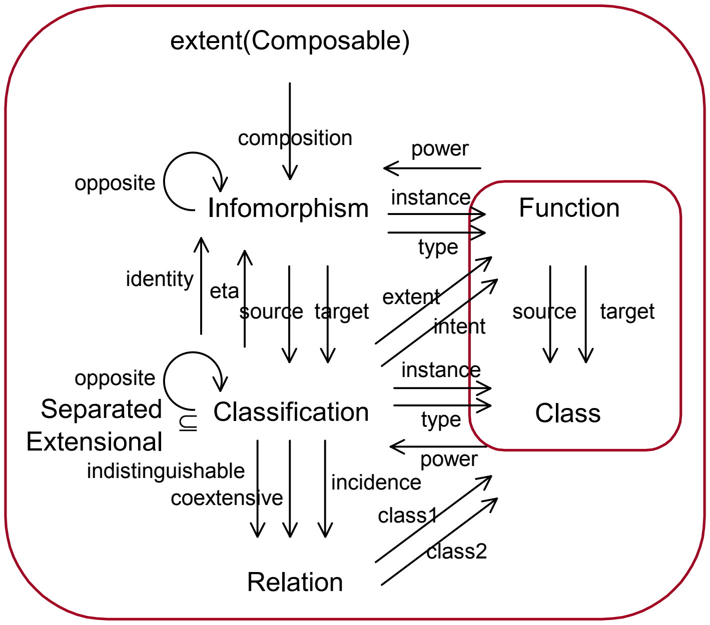


**Diagram 1: Core Collections and Functions for Classifications and Infomorphisms**

## *Classifications*

**CLS**

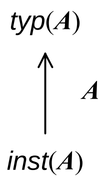


**Figure 1: Classification**

o A (large) *classification* $\boldsymbol{A} = \langle \mathit{inst}(\boldsymbol{A}), \mathit{typ}(\boldsymbol{A}), \models_{\boldsymbol{A}} \rangle$ (Figure 1) is identical to a (large) binary relation. However, from a category-theoretic standpoint, the context of classifications is very different from the context of relations, since their morphisms are very different. A large classification consists of a class of instances $\mathit{inst}(\boldsymbol{A})$ identified with the first or source class of a binary relation, a class of types $\mathit{typ}(\boldsymbol{A})$ identified with the second or target class of a binary relation, and a class of incidence or classification $\models_{\boldsymbol{A}}$ identified with the extent class of a binary relation.

The following is a KIF representation for the elements of a classification. The elements in the KIF representation are useful for the specification of a classification by declaration and population. The term ‘`classification`’ allows one to *declare* classifications. The terms ‘`instance`’, ‘`type`’ and ‘`incidence`’ resolve classifications into their parts, thus allowing one to *populate* classifications.

```
(1) (KIF$collection classification)
    (= classification REL$relation)

(2) (KIF$function instance)
    (= (KIF$source instance) classification)
    (= (KIF$target instance) SET$class)
    (= instance REL$class1)

(3) (KIF$function type)
    (= (KIF$source type) classification)
    (= (KIF$target type) SET$class)
    (= type REL$class2)

(4) (KIF$function incidence)
    (= (KIF$source incidence) classification)
    (= (KIF$target incidence) SET$class)
    (= incidence REL$extent)
```

- Associated with any classification is a function that produces the *intent* of an instance and a function that produces the *extent* of a type, both within the context of the classification. Dually, the intent of an instance $i \in \mathit{inst}(\boldsymbol{A})$ in a classification $\boldsymbol{A} = \langle \mathit{inst}(\boldsymbol{A}), \mathit{typ}(\boldsymbol{A}), \vDash_{\boldsymbol{A}} \rangle$ is defined by

  $\mathit{intent}_{\boldsymbol{A}}(i) = \{t \in \mathit{typ}(\boldsymbol{A}) \mid i \vDash_{\boldsymbol{A}} t\}$.

  The extent of a type $t \in \mathit{typ}(\boldsymbol{A})$ in a classification $\boldsymbol{A}$ defined by

  $\mathit{extent}_{\boldsymbol{A}}(t) = \{i \in \mathit{inst}(\boldsymbol{A}) \mid i \vDash_{\boldsymbol{A}} t\}$.

  Intent and extent are synonymous with relational fibers (12 and 21, respectively). The following axioms specify the intent and extent functions.

```
(5) (KIF$function intent)
    (= (KIF$source intent) classification)
    (= (KIF$target intent) SET.FTN$function)
    (= intent REL$fiber12)

(6) (KIF$function extent)
    (= (KIF$source extent) classification)
    (= (KIF$target extent) SET.FTN$function)
    (= extent REL$fiber21)
```

  These axioms demonstrate that the relative instantiation-predication represented by the incidence relation is compatible with, generalizes and relativizes the absolute KIF instantiation-predication – an instance is a member of the extent of a type (or dually, a type is a member of the intent of an instance) iff the instance is classified by the type.

```
    (forall (?a (classification ?a))
        (and (= (SET.FTN$source (extent ?a)) (type ?a))
             (= (SET.FTN$target (extent ?a)) (SET$power (instance ?a)))
             (forall (?i ((instance ?a) ?i) ?t ((type ?a) ?t))
                 (<=> (((extent ?a) ?t) ?i) (?a ?i ?t)))
             (= (SET.FTN$source (intent ?a)) (instance ?a))
             (= (SET.FTN$target (intent ?a)) (SET$power (type ?a)))
             (forall (?i ((instance ?a) ?i) ?t ((type ?a) ?t))
                 (<=> (((intent ?a) ?i) ?t) (?a ?i ?t)))))
```

- For any classification $\boldsymbol{A} = \langle \mathit{inst}(\boldsymbol{A}), \mathit{typ}(\boldsymbol{A}), \vDash_{\boldsymbol{A}} \rangle$, two instances $i_1, i_2 \in \mathit{inst}(\boldsymbol{A})$ are *indistinguishable* in $\boldsymbol{A}$ (Barwise and Seligman, 1997), written symbolically as $i_1 \sim_{\boldsymbol{A}} i_2$, when $\mathit{intent}_{\boldsymbol{A}}(i_1) = \mathit{intent}_{\boldsymbol{A}}(i_2)$. Two types $t_1, t_2 \in \mathit{typ}(\boldsymbol{A})$ are *coextensive* in $\boldsymbol{A}$, written symbolically as $t_1 \sim_{\boldsymbol{A}} t_2$, when $\mathit{extent}_{\boldsymbol{A}}(t_1) = \mathit{extent}_{\boldsymbol{A}}(t_2)$. A classification A is *separated* when there are no two distinct but indistinguishable instances, and *extensional* when there are no distinct coextensive types.

  The terms '`(indistinguishable ?a)`' and '`(coextensive ?a)`' represent the Information Flow notions of instance *indistinguishability* and type *coextension*, respectively. The terms '`separated`' and '`extensional`' represent the Information Flow notions of classification *separateness* and *extensionality*, respectively.

```
(7) (KIF$function indistinguishable)
    (= (KIF$source indistinguishable) classification)
    (= (KIF$target indistinguishable) REL.ENDO$relation)
    (forall (?a (classification ?a))
        (and (= (REL$class (indistinguishable ?a)) (instance ?a))
             (forall (?i1 ((instance ?a) ?i1)
                      ?i2 ((instance ?a) ?i2))
                 (<=> ((indistinguishable ?a) ?i1 ?i2)
                      (= ((intent ?a) ?i1) ((intent ?a) ?i2))))))

(8) (KIF$function coextensive)
    (= (KIF$source coextensive) classification)
    (= (KIF$target coextensive) REL.ENDO$relation)
    (forall (?a (classification ?a))
        (and (= (REL$class (coextensive ?a)) (type ?a))
             (forall (?t1 ((type ?a) ?t1)
                      ?t2 ((type ?a) ?t2))
                 (<=> ((coextensive ?a) ?t1 ?t2)
                      (= ((extent ?a) ?t1) ((extent ?a) ?t2))))))
```

```
(9) (KIF$collection separated)
    (KIF$subcollection separated classification)
    (forall (?a (classification ?a))
        (<=> (separated ?a)
             (REL.ENDO$subendorelation
                 (indistinguishable ?a)
                 (REL.ENDO$identity (instance ?a)))))

(10) (KIF$collection extensional)
     (KIF$subcollection extensional classification)
     (forall (?a (classification ?a))
         (<=> (extensional ?a)
              (REL.ENDO$subendorelation
                  (coextensive ?a)
                  (REL.ENDO$identity (type ?a)))))
```

- To quote (Barwise and Seligman, 1997), “in any classification, we think of the types as classifying the instances, but it is often useful to think of the instances as classifying the types.” For any classification $\boldsymbol{A} = \langle \mathit{inst}(\boldsymbol{A}), \mathit{typ}(\boldsymbol{A}), \models_{\boldsymbol{A}} \rangle$, the *opposite* or *dual* of $\boldsymbol{A}$ is the opposite binary relation; that is, the classification $\boldsymbol{A}^{\perp} = \langle \mathit{typ}(\boldsymbol{A}), \mathit{inst}(\boldsymbol{A}), \models_{\boldsymbol{A}}^{\perp} \rangle$, whose instances are types of $\boldsymbol{A}$, and whose types are instances of $\boldsymbol{A}$, and whose incidence is: $t \models^{\perp} i$ when $i \models t$.

```
(11) (KIF$function opposite)
     (= (KIF$source opposite) classification)
     (= (KIF$target opposite) classification)
     (= opposite REL$opposite)
```

- For any class $A$ the *instance power classification* $\wp A = \langle A, \wp A, \in_A \rangle$ over $A$ is defined as follows: the instance class is $A$; the type class is the power class $\wp A$ (so that a type is a subclass of $A$), and incidence is the membership relation $\in_A$.

```
(12) (KIF$function instance-power)
     (= (KIF$source instance-power) SET$class)
     (= (KIF$target instance-power) classification)
     (forall (?a (SET$class ?a))
         (and (= (instance (instance-power ?a)) ?a)
              (= (type (instance-power ?a)) (SET$power ?a))
              (forall (?x (?a ?x) ?y ((SET$power ?a) ?y))
                  (<=> ((instance-power ?a) ?x ?y) (?y ?x)))))
```

## Concept Lattices

**CLS.CL**

The central notions that are axiomatized in this section, are illustrated as an integrated framework in the partially commutative Diagram 2.

$$\begin{tikzcd}
& \mathit{concept}(\boldsymbol{A}) \arrow[dl, shift left, "\mathit{extent}_A"] \arrow[dr, shift left, "\mathit{intent}_A"] & \\
\wp(\mathit{inst}(\boldsymbol{A})) \arrow[ur, shift left, "\mathit{inst\text{-}gen}_A"] \arrow[rr, shift left, "\mathit{left\text{-}deriv}_A"] \arrow[loop left, "\mathit{inst\text{-}clsr}_A"] & & \wp(\mathit{typ}(\boldsymbol{A})) \arrow[ul, shift left, "\mathit{typ\text{-}gen}_A"] \arrow[ll, shift left, "\mathit{right\text{-}deriv}_A"] \arrow[loop right, "\mathit{typ\text{-}clsr}_A"]
\end{tikzcd}$$

**Diagram 2: Core classes/functions for Concept Lattices**

- For any large classification $\boldsymbol{A} = \langle \mathit{inst}(\boldsymbol{A}), \mathit{typ}(\boldsymbol{A}), \models_{\boldsymbol{A}} \rangle$ there are two senses of *derivation* corresponding to the two senses of relational residuation. Left derivation maps a subset of instances to the types on which they are all incident, and dually right derivation maps a subset of types to the instances, which are incident on all of them. For all $X \subseteq \mathit{inst}(\boldsymbol{A})$ and $Y \subseteq \mathit{typ}(\boldsymbol{A})$

  $X \mapsto X' = X \backslash \boldsymbol{A} = \{t \in \mathit{typ}(\boldsymbol{A}) \mid i \models_{\boldsymbol{A}} t \text{ for all } i \in X\}$

  $Y \mapsto Y' = \boldsymbol{A}/Y = \{i \in \mathit{inst}(\boldsymbol{A}) \mid i \models_{\boldsymbol{A}} t \text{ for all } t \in Y\}$.

```
(1) (KIF$function left-derivation)
    (= (KIF$source left-derivation) CLS$classification)
    (= (KIF$target left-derivation) SET.FTN$function)
    (forall (?a (CLS$classification ?a))
        (and (= (SET.FTN$source (left-derivation ?a)) (SET$power (CLS$instance ?a)))
             (= (SET.FTN$target (left-derivation ?a)) (SET$power (CLS$type ?a)))
             (forall (?x (SET$subclass ?x (CLS$instance ?a)))
                      ?t ((CLS$type ?a) ?t))
```

```
                  (<=> (((left-derivation ?a) ?x) ?t)
                       (forall (?i (?x ?i)) (?a ?i ?t))))))

(2) (KIF$function right-derivation)
    (= (KIF$source right-derivation) CLS$classification)
    (= (KIF$target right-derivation) SET.FTN$function)
    (forall (?a (CLS$classification ?a))
        (and (= (SET.FTN$source (right-derivation ?a)) (SET$power (CLS$type ?a)))
             (= (SET.FTN$target (right-derivation ?a)) (SET$power (CLS$instance ?a)))
             (forall (?y (SET$subclass ?y (CLS$type ?a)))
                     ?i ((CLS$instance ?a) ?i))
                 (<=> (((right-derivation ?a) ?y) ?i)
                      (forall (?t (?y ?t)) (?a ?I ?t))))))
```

○ A simple fundamental result is the following equivalence. For all $X \subseteq inst(\mathbf{A})$ and $Y \subseteq typ(\mathbf{A})$

$Y \subseteq X \backslash \mathbf{A}$ in $\wp(typ(\mathbf{A}))^{op}$ <u>iff</u> $X \times Y \subseteq \models_{\mathbf{A}}$ <u>iff</u> $X \subseteq \mathbf{A}/Y$ in $\wp(inst(\mathbf{A}))$.

This describes a Galois connection (preorder adjunction) between the left derivation

$(\text{-})\backslash\mathbf{A} : \wp(inst(\mathbf{A})) \rightarrow \wp(typ(\mathbf{A}))^{op}$

and the right derivation

$\mathbf{A}/(\text{-}) : \wp(typ(\mathbf{A}))^{op} \rightarrow \wp(inst(\mathbf{A}))$.

The first two facts assert (contravariant) monotonicity of derivation. The last fact asserts the adjointness condition.

```
    (forall (?x1 (SET$subclass ?x1 (CLS$instance ?a))
             ?x2 (SET$subclass ?x2 (CLS$instance ?a)))
        (=> (SET$subclass ?x1 ?x2)
            (SET$subclass ((left-derivation ?a) ?x2) ((left-derivation ?a) ?x1))))

    (forall (?y1 (SET$subclass ?y1 (CLS$type ?a))
             ?y2 (SET$subclass ?y2 (CLS$type ?a)))
        (=> (SET$subclass ?y2 ?y1)
            (SET$subclass ((right-derivation ?a) ?y1) ((right-derivation ?a) ?y2))))

    (forall (?x (SET$subclass ?x (CLS$instance ?a))
             ?y (SET$subclass ?y (CLS$type ?a)))
        (<=> (SET$subclass ?y ((left-derivation ?a) ?x))
             (SET$subclass ?x ((right-derivation ?a) ?y))))
```

○ The composition of derivations gives two senses of *closure* operator.

```
(3) (KIF$function instance-closure)
    (= (KIF$source instance-closure) CLS$classification)
    (= (KIF$target instance-closure) SET.FTN$function)
    (forall (?a (CLS$classification ?a))
        (and (= (SET.FTN$source (instance-closure ?a)) (SET$power (CLS$instance ?a)))
             (= (SET.FTN$target (instance-closure ?a)) (SET$power (CLS$instance ?a)))
             (= (instance-closure ?a)
                (SET.FTN$composition [(left-derivation ?a) (right-derivation ?a)]))))

(4) (KIF$function type-closure)
    (= (KIF$source type-closure) CLS$classification)
    (= (KIF$target type-closure) SET.FTN$function)
    (forall (?a (CLS$classification ?a))
        (and (= (SET.FTN$source (type-closure ?a)) (SET$power (CLS$type ?a)))
             (= (SET.FTN$target (type-closure ?a)) (SET$power (CLS$type ?a)))
             (= (type-closure ?a)
                (SET.FTN$composition [(right-derivation ?a) (left-derivation ?a)]))))
```

○ The following (easily proven) results confirm that these are closure operators.

$X \subseteq X''$ and $X' = X'''$ for all $X \subseteq inst(\mathbf{A})$.

$Y \subseteq Y''$ and $Y' = Y'''$ for all $Y \subseteq typ(\mathbf{A})$.

```
    (forall (?a (CLS$classification ?a))
        (and (= (left-derivation ?a)
                (SET.FTN$composition [(instance-closure ?a) (left-derivation ?a)]))
             (forall (?x (SET$subclass ?x (CLS$instance ?a))
                 (SET$subclass ?x ((instance-closure ?a) ?x)))))

    (forall (?a (CLS$classification ?a))
        (and (= (right-derivation ?a)
                (SET.FTN$composition [(type-closure ?a) (right-derivation ?a)]))
             (forall (?y (SET$subclass ?y (CLS$type ?a))
                 (SET$subclass ?y ((type-closure ?a) ?y)))))
```

- Further properties of Galois connection relate to continuity – the closure of a union of a family of classes is the intersection of the closures.

  $(\cup_{j\in J} X_j)' = \cap_{j\in J} X_j'$ for any family of subsets $X_j \subseteq \mathit{inst}(A)$ for $j\in J$.

  $(\cup_{k\in K} Y_k)' = \cap_{k\in K} Y_k'$ for any family of subsets $Y_k \subseteq \mathit{typ}(A)$ for $k\in K$.

- By embedding the subsets of instances and types above as relations, we can connect derivation to residuation. We can prove the following simple identities: the embedding of the left-derivation of a subset of instances is the left-residuation of the classification along the opposite of the embedding of the subset, and dually for the right notions.

```
(forall (?a (CLS$classification ?a))
         ?x (SET$subclass ?x (CLS$instance ?a)))
    (= ((REL$embed (CLS$type ?a)) ((left-derivation ?a) ?x))
       (REL$left-residuation [(REL$opposite ((REL$embed (CLS$instance ?a)) ?x)) ?a])))

(forall (?a (CLS$classification ?a))
         ?y (SET$subclass ?y (CLS$type ?a)))
    (= (REL$opposite ((REL$embed (CLS$instance ?a)) ((right-derivation ?a) ?y)))
       (REL$right-residuation [((REL$embed (CLS$type ?a)) ?y) ?a])))
```

- For any classification $A = \langle \mathit{inst}(A), \mathit{typ}(A), \vDash_A\rangle$, the closed elements of the derivation Galois connection are called *formal concepts*. There are several ways to define this notion, but traditionally it has been given the following (slightly redundant) definition. A (large) *formal concept* $c = \langle \mathit{extent}_A(c), \mathit{intent}_A(c)\rangle$ is a pair of classes, $\mathit{extent}_A(c) \subseteq \mathit{inst}(A)$ and $\mathit{intent}_A(c) \subseteq \mathit{typ}(A)$, that satisfy the equivalent conditions that $\mathit{extent}_A(c) = \mathit{intent}_A(c)'$ and $\mathit{intent}_A(c) = \mathit{extent}_A(c)'$. Let $\mathit{concept}(A)$ denote the class of all formal concepts of a classification $A$. There are two functions,

  $\mathit{extent}_A : \mathit{concept}(A) \rightarrow \wp(\mathit{inst}(A))$ and $\mathit{intent}_A : \mathit{concept}(A) \rightarrow \wp(\mathit{typ}(A))$,

  that map concepts to their component extent and intent.

```
(5) (KIF$function concept)
    (= (KIF$source concept) CLS$classification)
    (= (KIF$target concept) SET$class)

(6) (KIF$function extent)
    (= (KIF$source extent) CLS$classification)
    (= (KIF$target extent) SET.FTN$function)
    (forall (?a (CLS$classification ?a))
        (and (= (SET.FTN$source (extent ?a)) (concept ?a))
             (= (SET.FTN$target (extent ?a)) (SET$power (CLS$instance ?a)))))

(7) (KIF$function intent)
    (= (KIF$source intent) CLS$classification)
    (= (KIF$target intent) SET.FTN$function)
    (forall (?a (CLS$classification ?a))
        (and (= (SET.FTN$source (intent ?a)) (concept ?a))
             (= (SET.FTN$target (intent ?a)) (SET$power (CLS$type ?a)))))

(8) (forall (?a (CLS$classification ?a))
       (and (= (SET.FTN$composition [(intent ?a) (right-derivation ?a)]) (extent ?a))
            (= (SET.FTN$composition [(extent ?a) (left-derivation ?a)]) (intent ?a))))
```

○ There are surjective generator functions, called instance-generation and type-generation, that map subsets of instances and types to their generated concepts. For all $X \subseteq \mathit{inst}(\boldsymbol{A})$ and $Y \subseteq \mathit{typ}(\boldsymbol{A})$

$X \mapsto \langle X'', X' \rangle$ "instance-generation"

$Y \mapsto \langle Y', Y'' \rangle$ "type-generation".

See diagram 4 for a visualization of the following conditions.

– The composition of instance-generation and intent equals left-derivation, and the composition of type-generation and extent equals right-derivation.
– The composition of instance-generation and extent equals instance-closure, and the composition of type-generation and intent equals type-closure.
– The composition of extent and instance-generation is identity, and the composition of intent and type-generation is identity.

These conditions define generation, and also insure that all concepts are captured in the class '`(concept ?a)`'. Concepts are determined by either their extents or their intents – this is represented by the fact that the extent and intent functions are injective.

```
(9) (KIF$function instance-generation)
    (= (KIF$source instance-generation) CLS$classification)
    (= (KIF$target instance-generation) SET.FTN$function)
    (forall (?a (CLS$classification ?a))
        (and (= (SET.FTN$source (instance-generation ?a))
                (SET$power (CLS$instance ?a)))
             (= (SET.FTN$target (instance-generation ?a))
                (concept ?a))
             (= (SET.FTN$composition [(instance-generation ?a) (intent ?a)])
                (left-derivation ?a))
             (= (SET.FTN$composition [(instance-generation ?a) (extent ?a)])
                (instance-closure ?a))
             (= (SET.FTN$composition [(extent ?a) (instance-generation ?a)])
                (SET.FTN$identity (concept ?a)))))

(10) (KIF$function type-generation)
     (= (KIF$source type-generation) CLS$classification)
     (= (KIF$target type-generation) SET.FTN$function)
     (forall (?a (CLS$classification ?a))
         (and (= (SET.FTN$source (type-generation ?a))
                 (SET$power (CLS$type ?a)))
              (= (SET.FTN$target (type-generation ?a))
                 (concept ?a))
              (= (SET.FTN$composition [(type-generation ?a) (extent ?a)])
                 (right-derivation ?a))
              (= (SET.FTN$composition [(type-generation ?a) (intent ?a)])
                 (type-closure ?a))
              (= (SET.FTN$composition [(intent ?a) (type-generation ?a)])
                 (SET.FTN$identity (concept ?a)))))
```

○ A concept $c_1 = \langle \mathit{extent}_{\boldsymbol{A}}(c_1), \mathit{intent}_{\boldsymbol{A}}(c_1) \rangle$ is a *subconcept* of a concept $c_2 = \langle \mathit{extent}_{\boldsymbol{A}}(c_2), \mathit{intent}_{\boldsymbol{A}}(c_2) \rangle$, denoted $c_1 \leq_{\boldsymbol{A}} c_2$, when $\mathit{extent}_{\boldsymbol{A}}(c_1) \subseteq \mathit{extent}_{\boldsymbol{A}}(c_2)$ or equivalently when $\mathit{intent}_{\boldsymbol{A}}(c_1) \supseteq \mathit{intent}_{\boldsymbol{A}}(c_2)$. Other language used for this notion is that $c_1$ is "more specific" than $c_2$ and $c_2$ is "more generic" than $c_1$. For any classification $\boldsymbol{A} = \langle \mathit{inst}(\boldsymbol{A}), \mathit{typ}(\boldsymbol{A}), \models_{\boldsymbol{A}} \rangle$, the class of concepts together with the subconcept relation form a partial order $\mathit{ord}(\boldsymbol{A}) = \langle \mathit{concept}(\boldsymbol{A}), \leq_{\boldsymbol{A}} \rangle$.

```
(11) (KIF$function concept-order)
     (= (KIF$source concept-order) CLS$classification)
     (= (KIF$target concept-order) ORD$partial-order)
     (forall (?a (CLS$classification ?a))
         (and (= (ORD$class (concept-order ?a)) (concept ?a))
              (forall (?c1 ((concept ?a) ?c1)
                       ?c2 ((concept ?a) ?c2))
                  (<=> ((concept-order ?a) ?c1 ?c2)
                       (SET$subclass ((extent ?a) ?c1) ((extent ?a) ?c2))))))
```

○ There are both a meet and a join operations defined on subclasses of concepts

$$\mathit{meet}_{\boldsymbol{A}} : \wp(\mathit{concept}(\boldsymbol{A})) \rightarrow \mathit{concept}(\boldsymbol{A})$$

$$join_A : \wp(concept(A)) \rightarrow concept(A)$$

defined as follows:

$$\sqcap_L(C) = \sqcap_{c \in C} \langle extent_A(c), intent_A(c) \rangle = \langle \cap_{c \in C} extent_A(c), (\cup_{c \in C} intent_A(c))'' \rangle$$

$$\sqcup_L(C) = \sqcup_{c \in C} \langle extent_A(c), intent_A(c) \rangle = \langle (\cup_{c \in C} extent_A(c))'', \cap_{c \in C} intent_A(c) \rangle$$

for any subclass $C \subseteq concept(A)$.

```
(12) (KIF$function meet)
     (= (KIF$source meet) CLS$classification)
     (= (KIF$target meet) SET.FTN$function)
     (forall (?a (CLS$classification ?a))
         (and (= (SET.FTN$source (meet ?a)) (SET$power (concept ?a)))
              (= (SET.FTN$target (meet ?a)) (concept ?a))
              (forall (?c (SET$subclass ?c (concept ?a)))
                  (and (= (SET.FTN$composition [(meet ?a) (extent ?a)])
                          (SET.FTN$composition
                              [(SET.FTN$power (extent ?a))
                               (SET$intersection (instance ?a))]))
                       (= (SET.FTN$composition [(meet ?a) (intent ?a)])
                          (SET.FTN$composition
                              [(SET.FTN$composition
                                  [(SET.FTN$power (intent ?a))
                                   (SET$union (type ?a))])
                               (type-closure ?a)]))))))

(13) (KIF$function join)
     (= (KIF$source join) CLS$classification)
     (= (KIF$target join) SET.FTN$function)
     (forall (?a (CLS$classification ?a))
         (and (= (SET.FTN$source (join ?a)) (SET$power (concept ?a)))
              (= (SET.FTN$target (join ?a)) (concept ?a))
              (forall (?c (SET$subclass ?c (concept ?a)))
                  (and (= (SET.FTN$composition [(join ?a) (extent ?a)])
                          (SET.FTN$composition
                              [(SET.FTN$composition
                                  [(SET.FTN$power (extent ?a))
                                   (SET$union (instance ?a))])
                               (instance-closure ?a)]))
                       (= (SET.FTN$composition [(join ?a) (intent ?a)])
                          (SET.FTN$composition
                              [(SET.FTN$power (intent ?a))
                               (SET$intersection (type ?a))]))))))
```

- For any classification $A = \langle inst(A), typ(A), \vDash_A \rangle$, the partial order $order(A) = \langle concept(A), \leq_A \rangle$ of concepts together with the conceptual join and meet operators form a complete lattice $lat(A) = \langle order(A), join(A), meet(A) \rangle$.

The complete lattice associated with a classification is

– the image of the object function of the complete adjoint functor applied to the classification:

$$A : \text{BOND} \rightarrow \text{COMPLETE ADJOINT}.$$

– the image of the object function of the complete lattice functor applied to the classification:

$$A^2 : \text{BONDING PAIR} \rightarrow \text{COMPLETE LATTICE}.$$

```
(14) (KIF$function complete-lattice)
     (= (KIF$source complete-lattice) CLS$classification)
     (= (KIF$target complete-lattice) LAT$complete-lattice)
     (forall (?a (CLS$classification ?a))
         (and (= (LAT$partial-order (complete-lattice ?a)) (concept-order ?a))
              (= (LAT$join (complete-lattice ?a)) (join ?a))
              (= (LAT$meet (complete-lattice ?a)) (meet ?a))))
```

$$\begin{tikzcd}
\mathit{lat}(\boldsymbol{A}) \arrow[r, tail, "\tilde{\partial}_0"] \arrow[d, dash, "\leq_{\boldsymbol{A}}"'] \arrow[dr, dash] & \mathit{lat}(\boldsymbol{C}) \arrow[d, dash, "\leq_{\boldsymbol{C}}"] \\
\mathit{lat}(\boldsymbol{A}) & \mathit{lat}(\boldsymbol{C}) \arrow[l, hook', "\partial_0"]
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathit{lat}(\boldsymbol{C}) \arrow[r, hook, "\partial_1"] \arrow[d, dash, "\leq_{\boldsymbol{C}}"'] \arrow[dr, dash] & \mathit{lat}(\boldsymbol{B}) \arrow[d, dash, "\leq_{\boldsymbol{B}}"] \\
\mathit{lat}(\boldsymbol{C}) & \mathit{lat}(\boldsymbol{B}) \arrow[l, tail, "\tilde{\partial}_1"]
\end{tikzcd}$$

**Figure 2: Type-closed coreflection** **Figure 3: Instance-closed reflection**

- For any two classifications $\boldsymbol{A} = \langle \mathit{inst}(\boldsymbol{A}), \mathit{typ}, \vDash_{\boldsymbol{A}}\rangle$ and $\boldsymbol{C} = \langle \mathit{inst}(\boldsymbol{C}), \mathit{typ}, \vDash_{\boldsymbol{C}}\rangle$ that have the same class of types and where the classification (binary relation) $\boldsymbol{C}$ is *type-closed* $(\boldsymbol{A}/\boldsymbol{C})\backslash\boldsymbol{A} = \boldsymbol{C}$ with respect to $\boldsymbol{A}$ (Figure 2), there is an associated *coreflection* (adjoint pair) $\mathit{int}_{\boldsymbol{C},\boldsymbol{A}} = \langle\partial_0, \tilde{\partial}_0\rangle : \mathit{lat}(\boldsymbol{C}) \rightleftarrows \mathit{lat}(\boldsymbol{A})$ between their complete lattices, where $\tilde{\partial}_0 : \mathit{lat}(\boldsymbol{A}) \to \mathit{lat}(\boldsymbol{C})$ is right adjoint right inverse (rari) to $\partial_0 : \mathit{lat}(\boldsymbol{C}) \to \mathit{lat}(\boldsymbol{A})$. Hence, $\partial_0$ embeds $\mathit{lat}(\boldsymbol{C})$ as an internal part of $\mathit{lat}(\boldsymbol{A})$. These functions are defined as follows.

  $\partial_0(\langle X, Y\rangle) = (\langle Y', Y\rangle)$ for any concept $\langle X, Y\rangle \in \mathit{lat}(\boldsymbol{C})$

  $\tilde{\partial}_0(\langle X, Y\rangle) = (\langle Y', Y''\rangle)$ for any concept $\langle X, Y\rangle \in \mathit{lat}(\boldsymbol{A})$

- Dually, for any two classifications $\boldsymbol{C} = \langle \mathit{inst}, \mathit{typ}(\boldsymbol{C}), \vDash_{\boldsymbol{C}}\rangle$ and $\boldsymbol{B} = \langle \mathit{inst}, \mathit{typ}(\boldsymbol{B}), \vDash_{\boldsymbol{B}}\rangle$ that have the same class of instances and where the classification (binary relation) $\boldsymbol{C}$ is *instance-closed* $\boldsymbol{B}/(\boldsymbol{C}\backslash\boldsymbol{B}) = \boldsymbol{C}$ with respect to $\boldsymbol{B}$ (Figure 3), there is an associated *reflection* (adjoint pair) $\mathit{ext}_{\boldsymbol{B},\boldsymbol{C}} = \langle\tilde{\partial}_1, \partial_1\rangle : \mathit{lat}(\boldsymbol{B}) \rightleftarrows \mathit{lat}(\boldsymbol{C})$ between their complete lattices, where $\tilde{\partial}_1 : \mathit{lat}(\boldsymbol{B}) \to \mathit{lat}(\boldsymbol{C})$ is left adjoint right inverse (lari) to $\partial_1 : \mathit{lat}(\boldsymbol{C}) \to \mathit{lat}(\boldsymbol{B})$. Hence, $\partial_1$ embeds $\mathit{lat}(\boldsymbol{C})$ as an external part of $\mathit{lat}(\boldsymbol{A})$. These functions are defined as follows.

  $\partial_1(\langle X, Y\rangle) = (\langle X, X'\rangle)$ for any concept $\langle X, Y\rangle \in \mathit{lat}(\boldsymbol{C})$

  $\tilde{\partial}_1(\langle X, Y\rangle) = (\langle X'', X'\rangle)$ for any concept $\langle X, Y\rangle \in \mathit{lat}(\boldsymbol{B})$.

```
(15) (KIF$relation type-closed)
     (= (KIF$collection1 type-closed) CLS$classification)
     (= (KIF$collection2 type-closed) CLS$classification)
     (forall (?c (CLS$classification ?c)
              ?a (CLS$classification ?a))
         (<=> (type-closed ?c ?a)
              (and (= (CLS$type ?c) (CLS$type ?a))
                   (= (REL$left-residuation [(REL$right-residuation [?c ?a]) ?a]) ?c))))

(16) (KIF$function coreflection)
     (= (KIF$source coreflection) (KIF$extent type-closed))
     (= (KIF$target coreflection) LAT.ADJ$adjoint-pair)
     (forall (?c (CLS$classification ?c)
              ?a (CLS$classification ?a) (type-closed ?c ?a))
        (and (= (LAT.ADJ$source (coreflection [?c ?a])) (complete-lattice c?))
             (= (LAT.ADJ$target (coreflection [?c ?a])) (complete-lattice ?a))
             (= (SET.FTN$composition [(LAT.ADJ$left (coreflection [?c ?a])) (extent ?a)])
                (SET.FTN$composition [(intent ?c) (right-derivation ?c)]))
             (= (SET.FTN$composition [(LAT.ADJ$left (coreflection [?c ?a])) (intent ?a)])
                (intent ?c))
             (= (SET.FTN$composition [(LAT.ADJ$right (coreflection [?c ?a])) (extent ?c)])
                (SET.FTN$composition [(intent ?a) (right-derivation ?a)]))
             (= (SET.FTN$composition [(LAT.ADJ$right (coreflection [?c ?a])) (intent ?c)])
                (SET.FTN$composition [(intent ?a) (type-closure ?a)]))))

(17) (KIF$relation instance-closed)
     (= (KIF$collection1 instance-closed) CLS$classification)
     (= (KIF$collection2 instance-closed) CLS$classification)
     (forall (?b (CLS$classification ?b)
              ?c (CLS$classification ?c))
         (<=> (instance-closed ?c ?b)
              (and (= (CLS$instance ?b) (CLS$instance ?c))
                   (= (REL$right-residuation [(REL$left-residuation [?c ?b]) ?b]) ?c))))
```

```
(18) (KIF$function reflection)
     (= (KIF$source reflection (KIF$extent instance-closed))
     (= (KIF$target reflection) LAT.ADJ$adjoint-pair)
     (forall (?c (CLS$classification ?c)
              ?b (CLS$classification ?b) (instance-closed ?c ?b))
        (and (= (LAT.ADJ$source (reflection [?b ?c])) (complete-lattice b?))
             (= (LAT.ADJ$target (reflection [?b ?c])) (complete-lattice c?))
             (= (REL.FTN$composition [(LAT.ADJ$right (reflection [?b ?c])) (extent ?b)])
                (extent ?c))
             (= (REL.FTN$composition [(LAT.ADJ$right (reflection [?b ?c])) (intent ?b)])
                (REL.FTN$composition [(extent ?c) (left-derivation c?)]))
             (= (REL.FTN$composition [(LAT.ADJ$left (reflection [?b ?c])) (extent ?c)])
                (REL.FTN$composition [(extent ?b) (instance-closure ?b)]))
             (= (REL.FTN$composition [(LAT.ADJ$left (reflection [?b ?c])) (intent ?c)])
                (REL.FTN$composition [(extent ?b) (left-derivation ?b)]))))
```

o For any classification $\boldsymbol{A} = \langle \mathit{inst}(\boldsymbol{A}), \mathit{typ}(\boldsymbol{A}), \models_{\boldsymbol{A}} \rangle$, we can restrict the instance-generation and type-generation to elements, thus defining an instance embedding function $\iota_{\boldsymbol{A}} : \mathit{inst}(\boldsymbol{A}) \rightarrow \mathit{lat}(\boldsymbol{A})$ and a type embedding function $\tau_{\boldsymbol{A}} : \mathit{typ}(\boldsymbol{A}) \rightarrow \mathit{lat}(\boldsymbol{A})$.

```
(19) (KIF$function instance-embedding)
     (= (KIF$source instance-embedding) CLS$classification)
     (= (KIF$target instance-embedding) SET.FTN$function)
     (forall (?a (CLS$classification ?a))
         (and (= (SET.FTN$source (instance-embedding ?a)) (CLS$instance ?a))
              (= (SET.FTN$target (instance-embedding ?a)) (concept ?a))
              (= (instance-embedding ?a)
                 (SET.FTN$composition
                     [(SET.FTN$singleton (CLS$instance ?a))
                      (instance-generation ?a)]))))
```

```
(20) (KIF$function type-embedding)
     (= (KIF$source type-embedding) CLS$classification)
     (= (KIF$target type-embedding) SET.FTN$function)
     (forall (?a (CLS$classification ?a))
         (and (= (SET.FTN$source (type-embedding ?a)) (CLS$type ?a))
              (= (SET.FTN$target (type-embedding ?a)) (concept ?a))
              (= (type-embedding ?a)
                 (SET.FTN$composition
                     [(SET.FTN$singleton (CLS$type ?a))
                      (type-generation ?a)]))))
```

○ By means of these two embedding functions, we can prove that the notions of extent and intent for classifications extends to the notions of extent and intent for concept lattices.

$\mathit{extent}_{\boldsymbol{A}}$ (for classifications) = $\tau_{\boldsymbol{A}} \cdot \mathit{extent}_{\boldsymbol{A}}$ (for concept lattices)

$\mathit{intent}_{\boldsymbol{A}}$ (for classifications) = $\iota_{\boldsymbol{A}} \cdot \mathit{intent}_{\boldsymbol{A}}$ (for concept lattices)

For clarity, we state these identities in an external namespace.

```
 (forall (?a (CLS$classification ?a))
     (and (= (CLS$extent ?a)
             (SET.FTN$composition [(CLS.CL$type-embedding ?a) (CLS.CL$extent ?a)]))
          (= (CLS$intent ?a)
             (SET.FTN$composition [(CLS.CL$instance-embedding ?a) (CLS.CL$intent ?a)]))))
```

o Concepts in $\iota_{\boldsymbol{A}}(\mathit{inst}(\boldsymbol{A}))$ are called *instance concepts*, whereas concepts in $\tau_{\boldsymbol{A}}(\mathit{typ}(\boldsymbol{A}))$ are called *type concepts*.

```
(21) (KIF$function instance-concept)
     (= (KIF$source instance-concept) CLS$classification)
     (= (KIF$target instance-concept) CLS$class)
     (forall (?a (CLS$classification ?a))
         (and (SET$subclass (instance-concept ?a) (concept ?a))
                (= (instance-concept ?a)
                   (SET.FTN$image (instance-embedding ?a)))))

(22) (KIF$function type-concept)
     (= (KIF$source type-concept) CLS$classification)
     (= (KIF$target type-concept) CLS$class)
     (forall (?a (CLS$classification ?a))
         (and (SET$subclass (type-concept ?a) (concept ?a))
              (= (type-concept ?a)
                 (SET.FTN$image (type-embedding ?a)))))
```

○ Because of the resolutions

$$c = \bigsqcup_{i \in extentA(c)} \iota_A(i) = \bigsqcap_{t \in intentA(c)} \tau_A(t)$$

for any concept $c \in$ *concept*($\boldsymbol{A}$), these functions satisfy the following conditions.

– The image $\iota_A$(*inst*($\boldsymbol{A}$)) is join-dense in *ord(A)*.
– The image $\tau_A$(*typ*($\boldsymbol{A}$)) is meet-dense in *ord(A)*.
– The classification incidence can be expressed in terms of embeddings and lattice order:

$$i \vDash_A t \text{ iff } \iota_A(i) \leq_A \tau_A(t).$$

```
     (forall (?a (CLS$classification ?a))
         (and ((ORD$join-dense (concept-order ?a)) (instance-concept ?a))
              ((ORD$meet-dense (concept-order ?a)) (type-concept ?a))
              (forall (?i ((instance ?a) ?i) ?t ((type ?a) ?t))
                  (<=> (?a ?i ?t)
                       ((concept-order ?a)
                           ((instance-embedding ?a) ?i) ((type-embedding ?a) ?t))))))
```

o For any classification $\boldsymbol{A}$ = ⟨*inst*($\boldsymbol{A}$), *typ*($\boldsymbol{A}$), $\vDash_A$⟩, there are two classifications (binary relations) that correspond to the instance and type embedding functions.

The *instance embedding classification* $\iota_A$ = ⟨*inst*($\boldsymbol{A}$), *concept*($\boldsymbol{A}$), $\iota_A$⟩ (Figure 4) is defined as follows: for every instance $a \in$ *inst*($\boldsymbol{A}$) and every formal concept $\boldsymbol{a} \in$ *concept*($\boldsymbol{A}$), the incidence relationship $a\iota_A\boldsymbol{a}$ holds when $a$ is in the extent of $\boldsymbol{a}$; that is, $a \in$ *ext*$_A$($\boldsymbol{a}$). As a relation, this classification is closed on the right with respect to lattice order. The instance embedding classification can be defined in terms of the instance embedding function as follows:

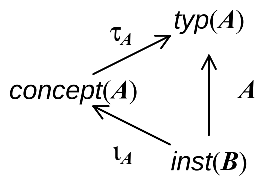


**Figure 4: *iota* & *tau***

$a\iota_A\boldsymbol{a}$ when $\iota_A(a) \leq_A \boldsymbol{a}$, for $a \in$ *inst*($\boldsymbol{A}$) and $\boldsymbol{a} \in$ *concept*($\boldsymbol{A}$).

This is an application of the right operator for a preorder that maps functions to binary relations.

The *type embedding classification* $\tau_A$ = ⟨*concept*($\boldsymbol{A}$), *typ*($\boldsymbol{A}$), $\tau_A$⟩ (Figure 4) is defined as follows: for every formal concept $\boldsymbol{a} \in$ *concept*($\boldsymbol{A}$) and every type $\alpha \in$ *typ*($\boldsymbol{A}$), the incidence relationship $\boldsymbol{a}\tau_A\alpha$ holds when $\alpha$ is in the intent of $\boldsymbol{a}$; that is, $\alpha \in$ *int*$_A$($\boldsymbol{a}$). As a relation, this classification is closed on the left with respect to lattice order. The type embedding classification can be defined in terms of the type embedding function as follows:

$\boldsymbol{a}\tau_A\alpha$ when $\boldsymbol{a} \leq_A \tau_A(\alpha)$, for $\boldsymbol{a} \in$ *concept*($\boldsymbol{A}$) and $\alpha \in$ *typ*($\boldsymbol{A}$).

This is an application of the left operator for a preorder that maps functions to binary relations.

```
(23) (KIF$function iota)
     (= (KIF$source iota) CLS$classification)
     (= (KIF$target iota) CLS$classification)
     (forall (?a (CLS$classification ?a))
         (and (= (CLS$instance (iota ?a)) (CLS$instance ?a))
              (= (CLS$type (iota ?a)) (concept ?a))
              (= (iota ?a)
                 (SET.FTN$right [(instance-embedding ?a) (concept-order ?a)]))))

(24) (KIF$function tau)
     (= (KIF$source tau) CLS$classification)
     (= (KIF$target tau) CLS$classification)
```

```
        (forall (?a (CLS$classification ?a))
            (and (= (CLS$instance (tau ?a)) (concept ?a))
                 (= (CLS$type (tau ?a)) (CLS$type ?a))
                 (= (tau ?a)
                    (SET.FTN$left [(type-embedding ?a) (concept-order ?a)]))))
```

o Conversely, the instance embedding function can be defined in terms of the instant embedding relation

$\iota_A(i) = \sqcap_A(i\iota_A)$, for $i \in$ *inst*($A$), and

the type embedding function can be defined in terms of the type embedding relation

$\tau_A(t) = \sqcup_A(\tau_A t)$, for $t \in$ *typ*($A$).

Since we have already defined these functions by other means, these facts are expressed as theorems in an external namespace.

```
        (forall (?a (CLS$classification ?a)) ?i ((CLS$instance ?a) ?i))
            (= ((CLS.CL$instance-embedding ?a) ?i)
               ((CLS.CL$meet ?a) ((CLS$intent (CLS.CL$iota ?a)) ?i))))

        (forall (?a (CLS$classification ?a)) ?t ((CLS$type ?a) ?t))
            (= ((CLS.CL$type-embedding ?a) ?t)
               ((CLS.CL$join ?a) ((CLS$extent (CLS.CL$tau ?a)) ?t))))
```

○ Intent induces a preorder on the instance class *inst*($A$) defined by $i_1 \preccurlyeq_A i_2$ when $\iota_A(i_1) \leq_A \iota_A(i_2)$ or $i_1' \supseteq i_2'$. Dually, extent induces a preorder on the type class *typ*($A$) defined by $t_1 \preccurlyeq_A t_2$ when $\tau_A(t_1) \leq_A \tau_A(t_2)$ or $t_1' \subseteq t_2'$.

```
(25) (KIF$function instance-order)
     (= (KIF$source instance-order) CLS$classification)
     (= (KIF$target instance-order) ORD$preorder)
     (forall (?a (CLS$classification ?a))
         (and (= (ORD$class (instance-order ?a)) (instance ?a))
              (forall (?i1 ((instance ?a) ?i1)
                       ?i2 ((instance ?a) ?i2))
              (<=> ((instance-order ?a) ?i1 ?i2)
                   ((concept-order ?a)
                       ((instance-embedding ?a) ?i1)
                       ((instance-embedding ?a) ?i2)))))

(26) (KIF$function type-order)
     (= (KIF$source type-order) CLS$classification)
     (= (KIF$target type-order) ORD$preorder)
     (forall (?a (CLS$classification ?a))
         (and (= (ORD$class (type-order ?a)) (type ?a))
              (forall (?t1 ((type ?a) ?t1)
                       ?t2 ((type ?a) ?t2))
              (<=> ((type-order ?a) ?t1 ?t2)
                   ((concept-order ?a)
                       ((type-embedding ?a) ?t1)
                       ((type-embedding ?a) ?t2)))))
```

○ Part of the fundamental theorem of Formal Concept Analysis states that every classification $A$ = ⟨*inst*($A$), *typ*($A$), $\models_A$⟩ has an associated concept lattice *cl*($A$) = ⟨*lat*($A$), *inst*($A$), *typ*($A$), $\iota_A$, $\tau_A$⟩.

The concept lattice associated with a classification is the image of the object function of the concept lattice functor applied to the classification:

$L$ : CLASSIFICATION → CONCEPT LATTICE.

```
(27) (KIF$function concept-lattice)
     (= (KIF$source concept-lattice) CLS$classification)
     (= (KIF$target concept-lattice) CL$concept-lattice)
     (forall (?a (CLS$classification ?a))
         (and (= (CL$complete-lattice (concept-lattice ?a)) (complete-lattice ?a))
              (= (CL$instance (concept-lattice ?a)) (CLS$instance ?a))
              (= (CL$type (concept-lattice ?a)) (CLS$type ?a))
```

```
                    (= (CL$instance-embedding (concept-lattice ?a)) (instance-embedding ?a))
                    (= (CL$type-embedding (concept-lattice ?a)) (type-embedding ?a))))
```

- The following fact of abstract Formal Concept Analysis, stated in terms of the embedding relations above, is straightforward to prove:

  $$\boldsymbol{A} = \iota_{\boldsymbol{A}} \circ \leq_{\boldsymbol{A}} \circ \tau_{\boldsymbol{A}}^{\mathrm{op}}.$$

  This says that any classification $\boldsymbol{A} = \langle \mathit{inst}(\boldsymbol{A}), \mathit{typ}(\boldsymbol{A}), \models_{\boldsymbol{A}}\rangle$ (viewed as a binary relation) is identical to the composition of the instance-embedding relation followed by the lattice order of the classification followed by the type embedding relation.

```
 (forall (?a (CLS$classification ?a))
     (= ?a
        (REL$composition [(REL$composition [(iota ?a) (concept-order ?a)]) (tau ?a)])))
```

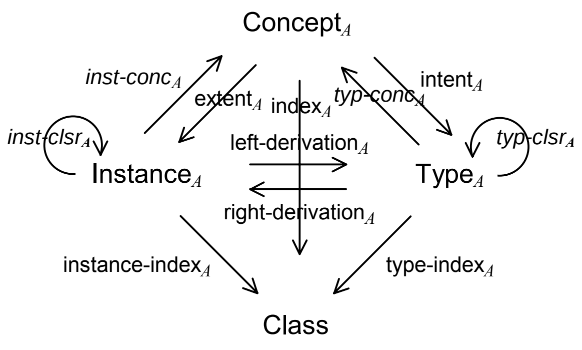


**Diagram 3: Core Collections and Functions for Fibers**

## *Conceptual Fibers*

**`CLS.FIB`**

This section defines fibers along the function

*classification* : Concept → CLASSIFICATION

or compositions with this function (see Diagram 4).

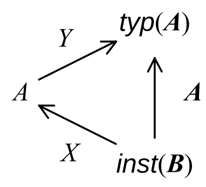


**Figure 5: *A*-instances and *A*-types**

- For any classification $\boldsymbol{A} = \langle \mathit{inst}(\boldsymbol{A}), \mathit{typ}(\boldsymbol{A}), \models_{\boldsymbol{A}}\rangle$, a (*collective*) ***A**-instance* (Figure 5) indexed by a class $A$ is a binary relation $X \subseteq \mathit{inst}(\boldsymbol{A}) \times A$. For a fixed classification $\boldsymbol{A}$, the collection of all $\boldsymbol{A}$-instances is denoted '`(instance ?a)`'. Dually, a (*collective*) ***A**-type* indexed by a class $A$ is a binary relation $Y \subseteq A \times \mathit{typ}(\boldsymbol{A})$. For a fixed classification $\boldsymbol{A}$, the collection of all $\boldsymbol{A}$-types is denoted '`(type ?a)`'.

```
(1) (KIF$function instance)
    (= (KIF$source instance) CLS$classification)
    (= (KIF$target instance) KIF$collection)
    (forall (?a (CLS$classification ?a)
        (and (KIF$subcollection (instance ?a) REL$relation)
             (forall (?x (REL$relation ?x))
                 (<=> ((instance ?a) ?x)
                      (= (REL$source ?x) (CLS$instance ?a))))))

(2) (KIF$function instance-index)
    (= (KIF$source instance-index) CLS$classification)
    (= (KIF$target instance-index) KIF$function)
    (forall (?a (CLS$classification ?a))
        (and (= (KIF$source (instance-index ?a)) (instance ?a))
             (= (KIF$target (instance-index ?a)) SET$class)
             (forall (?x ((instance ?a) ?x))
                 (= ((instance-index ?a) ?x) (REL$target ?x)))))

(3) (KIF$function type)
    (= (KIF$source type) CLS$classification)
    (= (KIF$target type) KIF$collection)
    (forall (?a (CLS$classification ?a)
        (and (KIF$subcollection (type ?a) REL$relation)
             (forall (?y (REL$relation ?y))
                 (<=> ((type ?a) ?y)
                      (= (REL$target ?y) (CLS$type ?a))))))

(4) (KIF$function type-index)
    (= (KIF$source type-index) CLS$classification)
    (= (KIF$target type-index) KIF$function)
    (forall (?a (CLS$classification ?a))
        (and (= (KIF$source (type-index ?a)) (type ?a))
             (= (KIF$target (type-index ?a)) SET$class)
             (forall (?y ((type ?a) ?y))
                 (= ((type-index ?a) ?y) (REL$source ?y)))))
```

○ Two derivation operators in this fibered setting correspond to the two relational residuation operators. For any large classification $\boldsymbol{A}$ there are two senses of *derivation* corresponding to the two senses of relational residuation. Derivation preserves indices. Left derivation maps an instance to the type on which it is "universally incident," and dually, right derivation maps a type to the instance which is "universally incident" on it: for all instances $X \subseteq \mathit{inst}(\boldsymbol{A}) \times A$ and all types $Y \subseteq A \times \mathit{typ}(\boldsymbol{A})$,

$X \mapsto X' = X \backslash \boldsymbol{A}$ and $Y \mapsto Y' = \boldsymbol{A}/Y$.

```
(5) (KIF$function left-derivation)
    (= (KIF$source left-derivation) CLS$classification)
    (= (KIF$target left-derivation) KIF$function)
    (forall (?a (CLS$classification ?a))
        (and (= (KIF$source (left-derivation ?a)) (instance ?a))
             (= (KIF$target (left-derivation ?a)) (type ?a))
             (forall (?x ((instance ?a) ?x))
                 (= ((left-derivation ?a) ?x)
                    (REL$left-residuation [?x ?a])))))

(6) (KIF$function right-derivation)
    (= (KIF$source right-derivation) CLS$classification)
    (= (KIF$target right-derivation) SET.FTN$function)
    (forall (?a (CLS$classification ?a))
        (and (= (KIF$source (right-derivation ?a)) (type ?a))
             (= (KIF$target (right-derivation ?a)) (instance ?a))
             (forall (?y ((type ?a) ?y))
                 (= ((right-derivation ?a) ?y)
                    (REL$right-residuation [?y ?a])))))
```

○ A simple result is the following equivalence. For all instances $X \subseteq \mathit{inst}(\boldsymbol{A}) \times A$ and types $Y \subseteq A \times \mathit{typ}(\boldsymbol{A})$

$Y \subseteq X \backslash \boldsymbol{A}$ in $\mathit{REL}[A, \mathit{typ}(\boldsymbol{A})]$ <u>iff</u> $X \circ Y \subseteq \boldsymbol{A}$ <u>iff</u> $X \subseteq \boldsymbol{A}/Y$ in $\mathit{REL}[\mathit{inst}(\boldsymbol{A}), A]$.

This describes a Galois connection between left derivation

$(\text{-})\backslash \boldsymbol{A} : \mathit{REL}[\mathit{inst}(\boldsymbol{A}), A] \rightarrow (\mathit{REL}[A, \mathit{typ}(\boldsymbol{A})])^{\mathrm{op}}$

and right derivation

$\boldsymbol{A}/(\text{-}) : (\mathit{REL}[A, \mathit{typ}(\boldsymbol{A})])^{\mathrm{op}} \rightarrow \mathit{REL}[\mathit{inst}(\boldsymbol{A}), A]$.

The first two facts assert (contravariant) monotonicity of derivation. The last fact asserts the adjointness condition.

```
    (forall (?x1 ((instance ?a) ?x1) ?x2 ((instance ?a) ?x2)
             (= ((instance-index ?a) ?x1) ((instance-index ?a) ?x2)))
        (=> (REL$subrelation ?x1 ?x2)
            (REL$subrelation ((left-derivation ?a) ?x2) ((left-derivation ?a) ?x1))))

    (forall (?y1 ((type ?a) ?y1) ?y2 ((type ?a) ?y2)
             (= ((type-index ?a) ?y1) ((type-index ?a) ?y2)))
        (=> (REL$subrelation ?y2 ?y1)
            (REL$subrelation ((right-derivation ?a) ?y1) ((right-derivation ?a) ?y2))))

    (forall (?x ((instance ?a) ?x) ?y ((type ?a) ?y)
             (= ((instance-index ?a) ?x) ((type-index ?a) ?y)))
        (<=> (REL$subrelation ?y ((left-derivation ?a) ?x))
             (REL$subrelation ?x ((right-derivation ?a) ?y))))
```

○ The composition of derivations gives two senses of closure operator.

```
(7) (KIF$function instance-closure)
    (= (KIF$source instance-closure) CLS$classification)
    (= (KIF$target instance-closure) KIF$function)
    (forall (?a (CLS$classification ?a))
        (and (= (KIF$source (instance-closure ?a)) (instance ?a))
             (= (KIF$target (instance-closure ?a)) (instance ?a))
             (forall (?x ((instance ?a) ?x))
                 (and (= (instance-index ((instance-closure ?a) ?x))
                         (instance-index ?x))
                      (= ((instance-closure ?a) ?x)
                         ((right-derivation ?a) ((left-derivation ?a) ?x)))))))

(8) (KIF$function type-closure)
    (= (KIF$source type-closure) CLS$classification)
    (= (KIF$target type-closure) KIF$function)
    (forall (?a (CLS$classification ?a))
        (and (= (KIF$source (type-closure ?a)) (type ?a))
             (= (KIF$target (type-closure ?a)) (type ?a))
             (forall (?y ((type ?a) ?y))
                 (and (= (type-index ((type-closure ?a) ?y))
                         (type-index ?y))
                      (= ((type-closure ?a) ?y)
                         ((left-derivation ?a) ((right-derivation ?a) ?y)))))))
```

○ The following (easily proven) results confirm that these are closure operators.

$X \subseteq X''$ and $X' = X'''$ for all instances $X \subseteq \mathit{inst}(\boldsymbol{A}) \times A$.

$Y \subseteq Y''$ and $Y' = Y'''$ for all types $Y \subseteq A \times \mathit{typ}(\boldsymbol{A})$.

```
    (forall (?x ((instance ?a) ?x))
        (and (REL$subrelation ?x ((instance-closure ?a) ?x))
             (= ((left-derivation ?a) ?x)
                ((left-derivation ?a) ((instance-closure ?a) ?x)))))

    (forall (?y ((type ?a) ?y))
        (and (REL$subrelation ?y ((type-closure ?a) ?y))
             (= ((right-derivation ?a) ?y)
                ((right-derivation ?a) ((type-closure ?a) ?y)))))
```

o For any classification $\boldsymbol{A} = \langle \mathit{inst}(\boldsymbol{A}), \mathit{typ}(\boldsymbol{A}), \vDash_{\boldsymbol{A}} \rangle$, a (*collective*) ***A**-concept* $\boldsymbol{C} = \langle \mathit{ind}(\boldsymbol{C}), \mathit{ext}(\boldsymbol{C}), \mathit{int}(\boldsymbol{C}) \rangle$ (Figure 6), consists of an *extent* $\boldsymbol{A}$-instance $\mathit{ext}(\boldsymbol{C}) : \mathit{inst}(\boldsymbol{A}) \rightarrow \mathit{ind}(\boldsymbol{A})$ indexed by $\mathit{ind}(\boldsymbol{A})$, and an *intent* $\boldsymbol{A}$-type $\mathit{int}(\boldsymbol{C}) : \mathit{ind}(\boldsymbol{A}) \rightarrow \mathit{typ}(\boldsymbol{A})$ indexed by $\mathit{ind}(\boldsymbol{A})$, which satisfy the following closure conditions:

$$\mathit{ext}(\boldsymbol{C}) = \boldsymbol{A}/\mathit{int}(\boldsymbol{C}) \text{ and } \mathit{int}(\boldsymbol{C}) = \mathit{ext}(\boldsymbol{C})\backslash\boldsymbol{A}.$$

$$\begin{tikzcd}
 & \mathit{typ}(\boldsymbol{A}) \\
\mathit{ind}(\boldsymbol{C}) \arrow[ur, "\mathit{int}(\boldsymbol{C})"] & \\
 & \mathit{inst}(\boldsymbol{A}) \arrow[ul, "\mathit{ext}(\boldsymbol{C})"] \arrow[uu, "\boldsymbol{A}"']
\end{tikzcd}$$

**Figure 6: *A*-Concept**

```
(9) (KIF$function concept)
    (= (KIF$source concept) CLS$classification)
    (= (KIF$target concept) KIF$collection)

(10) (KIF$function extent)
     (= (KIF$source extent) CLS$classification)
     (= (KIF$target extent) KIF$function)
     (forall (?a (CLS$classification ?a))
         (and (= (KIF$source (extent ?a)) (concept ?a))
              (= (KIF$target (extent ?a)) (instance ?a))))

(11) (KIF$function intent)
     (= (KIF$source intent) CLS$classification)
     (= (KIF$target intent) KIF$function)
     (forall (?a (CLS$classification ?a))
         (and (= (KIF$source (intent ?a)) (concept ?a))
              (= (KIF$target (intent ?a)) (type ?a))))

(12) (KIF$function index)
     (= (KIF$source index) CLS$classification)
     (= (KIF$target index) KIF$function)
     (forall (?a (CLS$classification ?a))
        (and (= (KIF$source (index ?a)) (concept ?a))
             (= (KIF$target (index ?a)) SET$class)
             (forall (?c ((concept ?a) ?c))
                 (and (= ((index ?a) ?c) ((instance-index ?a) ((extent ?a) ?c)))
                      (= ((index ?a) ?c) ((type-index ?a) ((intent ?a) ?c)))
                      (= ((left-derivation ?a) ((extent ?a) ?c)) ((intent ?a) ?c))
                      (= ((right-derivation ?a) ((intent ?a) ?c)) ((extent ?a) ?c))))))
```

○ There are surjective generator functions, called instance-generation and type-generation, that map instances and types to their generated concepts. For all instances $X \subseteq inst(A) \times A$ and types $Y \subseteq A \times typ(A)$

$X \mapsto \langle X'', X' \rangle$ “instance-generation”

$Y \mapsto \langle Y', Y'' \rangle$ “type-generation”.

– The composition of instance-generation and intent equals left-derivation, and the composition of type-generation and extent equals right-derivation.
– The composition of instance-generation and extent equals instance-closure, and the composition of type-generation and intent equals type-closure.
– The composition of extent and instance-generation is identity, and the composition of intent and type-generation is identity.

These conditions define generation, and also insure that all concepts are captured in the collection ‘`(concept ?a)`’. Concepts are determined by either their extents or their intents – this is represented by the fact that the extent and intent functions are injective.

```
(13) (KIF$function instance-generation)
     (= (KIF$source instance-generation) CLS$classification)
     (= (KIF$target instance-generation) KIF$function)
     (forall (?a (CLS$classification ?a))
         (and (= (KIF$source (instance-generation ?a)) (instance ?a))
              (= (KIF$target (instance-generation ?a)) (concept ?a))
              (forall (?x ((instance ?a) ?x))
                  (and (= ((index ?a) ((instance-generation ?a) ?x))
                          ((instance-index ?a) ?x))
                       (= ((intent ?a) ((instance-generation ?a) ?x))
                          ((left-derivation ?a) ?x))
                       (= ((extent ?a) ((instance-generation ?a) ?x))
                          ((instance-closure ?a) ?x))))
              (forall (?c ((concept ?a) ?c))
                  (= ((instance-generation ?a) ((extent ?a) ?c)) ?c))))

(14) (KIF$function type-generation)
     (= (KIF$source type-generation) CLS$classification)
     (= (KIF$target type-generation) KIF$function)
     (forall (?a (CLS$classification ?a))
         (and (= (KIF$source (type-generation ?a)) (type ?a))
              (= (KIF$target (type-generation ?a)) (concept ?a))
              (forall (?y ((type ?a) ?y))
                  (and (= ((index ?a) ((type-generation ?a) ?y))
                          ((type-index ?a) ?y))
                       (= ((extent ?a) ((type-generation ?a) ?y))
                          ((right-derivation ?a) ?y))
                       (= ((intent ?a) ((type-generation ?a) ?y))
                          ((type-closure ?a) ?y))))
              (forall (?c ((concept ?a) ?c))
                  (= ((type-generation ?a) ((intent ?a) ?c)) ?c))))
```

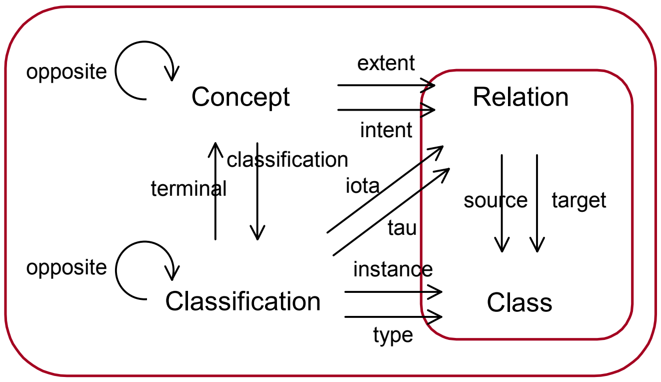


**Diagram 4: Core Collections and Functions for Concepts**

## Collective Concepts

**CLS.CONC**

Here we define concepts that internally include their underlying classifications.

$$\begin{tikzcd}
 & int(C) \arrow[r] & typ(cls(C)) \\
ind(C) \arrow[ur] & & cls(C) \arrow[u] \\
 & ext(C) & inst(cls(C)) \arrow[ull]
\end{tikzcd}$$

**Figure 7: Collective Concept**

o A formal (collective) concept $C = \langle cls(C), ind(C), ext(C), int(C)\rangle$ (Figure 7) consists of an underlying *classification* $cls(C)$, an *index*ing class $ind(C)$, an *extent* relation $ext(C)$ and an *intent* relation $int(C)$. These components satisfy the equivalent closure conditions:

$$ext(C) = cls(C)/int(C) \text{ and } int(C) = ext(C)\backslash cls(C).$$

The underlying classification function is part of a fibration. When the index function maps to unity (the unit class), the concept is a point. The collection of all concepts is the disjoint union of all the conceptual fibers $concept(A)$ as $A$ ranges over the collection of all large classifications. The last axiom expresses this.

```
(1) (KIF$collection concept)

(2) (KIF$function classification)
    (= (KIF$source classification) concept)
    (= (KIF$target classification) CLS$classification)

(3) (KIF$function index)
    (= (KIF$source index) concept)
    (= (KIF$target index) SET$class)

(4) (KIF$function extent)
    (= (KIF$source extent) concept)
    (= (KIF$target extent) REL$relation)
    (forall (?c (concept ?c))
        (and (= (REL$source (extent ?c)) (CLS$instance (classification ?c)))
             (= (REL$target (extent ?c)) (index ?c))))

(5) (KIF$function intent)
    (= (KIF$source intent) concept)
    (= (KIF$target intent) REL$relation)
    (forall (?c (concept ?c))
        (and (= (REL$source (intent ?c)) (index ?c))
             (= (REL$target (intent ?c)) (CLS$type (classification ?c)))))

(6) (forall (?c (concept ?c))
        (and (= (extent ?c) (REL$right-residuation [(intent ?c) (classification ?c)]))
             (= (intent ?c) (REL$left-residuation [(extent ?c) (classification ?c)]))))

(7) (forall (?c)
        (<=> (concept ?c)
             (exists (?cf ((CLS.FIB$concept (classfication ?c)) ?cf))
                 (and (= (index ?c) ((CLS.FIB$index (classfication ?c)) ?cf))
                      (= (extent ?c) ((CLS.FIB$extent (classfication ?c)) ?cf))
                      (= (intent ?c) ((CLS.FIB$intent (classfication ?c)) ?cf))))))
```

- For any concept $\boldsymbol{C} = \langle cls(\boldsymbol{C}), ind(\boldsymbol{C}), ext(\boldsymbol{C}), int(\boldsymbol{C})\rangle$, the *opposite* or *dual* of $\boldsymbol{C}$ is the concept $\boldsymbol{C}^{\perp} = \langle cls(\boldsymbol{C})^{\perp}, ind(\boldsymbol{C}), int(\boldsymbol{C}), ext(\boldsymbol{C})\rangle$, whose classification is the opposite of the classification of $\boldsymbol{C}$, whose index is the same as the index of $\boldsymbol{C}$, whose extent is the intent of $\boldsymbol{C}$, and whose intent is the extent of $\boldsymbol{C}$.

```
(8) (KIF$function opposite)
    (= (KIF$source opposite) concept)
    (= (KIF$target opposite) concept)
    (forall (?c (concept ?c))
        (and (= (classification (opposite ?c)) (CLS$opposite (classification ?c)))
             (= (index (opposite ?c)) (index ?c))
             (= (extent (opposite ?c)) (intent ?c))
             (= (intent (opposite ?c)) (extent ?c))))
```

- For any two collective concepts $\boldsymbol{C}_1$ and $\boldsymbol{C}_2$ over the same classification $cls(\boldsymbol{C}_1) = \boldsymbol{A} = cls(\boldsymbol{C}_2)$, a *two cell* $f : \boldsymbol{C}_1 \Rightarrow \boldsymbol{C}_2$ from $\boldsymbol{C}_1$ to $\boldsymbol{C}_2$ (Figure 8) consists of an *index function* $f : ind(\boldsymbol{C}_1) \to ind(\boldsymbol{C}_2)$ between index classes, which satisfies the following conditions:

$$ext(\boldsymbol{C}_1) = ext(\boldsymbol{C}_2) \circ f^{\text{op}} = ext(\boldsymbol{C}_2) \,/\, f$$

$$int(\boldsymbol{C}_1) = f \circ int(\boldsymbol{C}_2) = f^{\text{op}} \setminus int(\boldsymbol{C}_2)$$

$$\begin{tikzcd}
 & & typ(\boldsymbol{A}) \\
ind(\boldsymbol{C}_1) \arrow[r, "f"] \arrow[urr, "int(\boldsymbol{C}_1)"] & ind(\boldsymbol{C}_2) \arrow[ur, "int(\boldsymbol{C}_2)"'] & \\
 & & inst(\boldsymbol{A}) \arrow[uu, "\boldsymbol{A}"'] \arrow[ul, "ext(\boldsymbol{C}_2)"'] \arrow[ull, "ext(\boldsymbol{C}_1)"]
\end{tikzcd}$$

**Figure 8: 2-cell**

```
(9) (KIF$collection two-cell)

(10) (KIF$function source)
     (= (KIF$source source) two-cell)
     (= (KIF$target source) concept)

(11) (KIF$function target)
     (= (KIF$source target) two-cell)
     (= (KIF$target target) concept)

(12) (KIF$function index-function)
     (= (KIF$source index-function) two-cell)
     (= (KIF$target index-function) SET.FTN$function)

     (forall (?f (two-cell ?f))
         (and (= (classification (source ?f)) (classification (target ?f)))
              (= (index (source ?f)) (SET.FTN$source (index-function ?f)))
              (= (index (target ?f)) (SET.FTN$target (index-function ?f)))
              (= (REL$composition
                     [(extent (target ?f))
                      (REL$opposite (SET.FTN$fn2rel (index-function ?f)))])
                 (extent (source ?f)))
              (= (REL$composition
                     [(SET.FTN$fn2rel (index-function ?f))
                      (intent (target ?f))])
                 (intent (source ?f)))))
```

- Any classification $\boldsymbol{A}$ determines a *terminal* collective concept $terminal(\boldsymbol{A}) = \langle \boldsymbol{A}, concept(\boldsymbol{A}), \iota_{\boldsymbol{A}}, \tau_{\boldsymbol{A}}\rangle$. Its extent relation $\iota_{\boldsymbol{A}} \subseteq inst(\boldsymbol{A}) \times concept(\boldsymbol{A})$ is the instance embedding relation, its intent relation $\tau_{\boldsymbol{A}} \subseteq concept(\boldsymbol{A}) \times typ(\boldsymbol{A})$ is the type embedding relation, and its index is the concept class $concept(\boldsymbol{A})$. The closure conditions $\iota_{\boldsymbol{A}} = \boldsymbol{A}/\tau_{\boldsymbol{A}}$ and $\tau_{\boldsymbol{A}} = \iota_{\boldsymbol{A}} \setminus \boldsymbol{A}$ that hold between the instance embedding relation $\iota_{\boldsymbol{A}} : inst(\boldsymbol{A}) \to concept(\boldsymbol{A})$ and the type embedding relation $\tau_{\boldsymbol{A}} : concept(\boldsymbol{A}) \to typ(\boldsymbol{A})$ ensure that this is well-defined.

```
(13) (KIF$function terminal)
     (= (KIF$source terminal) CLS$classification)
     (= (KIF$target terminal) concept)
     (forall (?a (CLS$classification ?a))
         (and (= (classification (terminal ?a)) ?a)
              (= (index (terminal ?a)) (CLS.CL$concept ?a))
              (= (extent (terminal ?a)) (CLS.CL$iota ?a))
              (= (intent (terminal ?a)) (CLS.CL$tau ?a))))
```

○ Any collective concept $\boldsymbol{C} = \langle \boldsymbol{A}, A, X, Y\rangle = \langle \mathsf{cls}(\boldsymbol{C}), \mathsf{ind}(\boldsymbol{C}), \mathsf{ext}(\boldsymbol{C}), \mathsf{int}(\boldsymbol{C})\rangle$ induces a unique *mediating function* $\mu_{\boldsymbol{C}} : \mathsf{ind}(\boldsymbol{C}) \rightarrow \mathsf{concept}(\mathsf{cls}(\boldsymbol{C}))$ define pointwise by $\mu_{\boldsymbol{C}}(a) = \langle Xa, aY\rangle$ as $a$ ranges over the index class. This definition is well-defined, since the collective closure conditions $X = Y'$ and $Y = X'$ are equivalent to the (pointwise) closure conditions $Xa = (aY)'$ and $aY = (Xa)'$, as $a$ ranges over the index class; that is, $\langle Xa, aY\rangle \in \mathsf{concept}(\boldsymbol{A})$.

```
(14) (KIF$function mediator-function)
     (= (KIF$source mediator-function) concept)
     (= (KIF$target mediator-function) SET.FTN$function)
     (forall (?c (concept ?c))
         (and (= (SET.FTN$source (mediator-function ?c))
                 (index ?c))
              (= (SET.FTN$target (mediator-function ?c))
                 (CLS.CL$concept (classification ?c)))
              (= (SET.FTN$composition
                     [(mediator-function ?c)
                      (CLS.CL$extent (classification ?c))])
                 (REL$fiber21 (extent ?c)))
              (= (SET.FTN$composition
                     [(mediator-function ?c)
                      (CLS.CL$intent (classification ?c))])
                 (REL$fiber12 (intent ?c)))))
```

o Any collective concept $\boldsymbol{C} = \langle \boldsymbol{A}, A, X, Y\rangle = \langle \mathsf{cls}(\boldsymbol{C}), \mathsf{ind}(\boldsymbol{C}), \mathsf{ext}(\boldsymbol{C}), \mathsf{int}(\boldsymbol{C})\rangle$ induces a unique *mediator* 2-cell $\mu_{\boldsymbol{C}} : \boldsymbol{C} \Rightarrow \mathsf{terminal}(\boldsymbol{A}) = \mathsf{terminal}(\mathsf{cls}(\boldsymbol{C}))$ (Figure 9), since its mediator function satisfies the following constraints:

$$\mathsf{ext}(\boldsymbol{C}) = \iota_{\mathsf{cls}(\boldsymbol{C})} \circ \mu_{\boldsymbol{C}}^{\mathrm{op}} = \iota_{\mathsf{cls}(\boldsymbol{C})} / \mu_{\boldsymbol{C}}$$

$$\mathsf{int}(\boldsymbol{C}) = \mu_{\boldsymbol{C}} \circ \tau_{\mathsf{cls}(\boldsymbol{C})} = \mu_{\boldsymbol{C}}^{\mathrm{op}} \setminus \tau_{\mathsf{cls}(\boldsymbol{C})}.$$

$$\begin{tikzcd}
& & \mathsf{typ}(\mathsf{cls}(\boldsymbol{C})) \\
\mathsf{ind}(\boldsymbol{C}) \arrow[urr, "\mathsf{int}(\boldsymbol{C})"] \arrow[r, "\mu_{\boldsymbol{C}}"] & \mathsf{concept}(\mathsf{cls}(\boldsymbol{C})) \arrow[ur, "\tau_{\mathsf{cls}(\boldsymbol{C})}"] & \\
& & \mathsf{inst}(\mathsf{cls}(\boldsymbol{C})) \arrow[ul, "\iota_{\mathsf{cls}(\boldsymbol{C})}"'] \arrow[ull, "\mathsf{ext}(\boldsymbol{C})"] \arrow[uu, "\mathsf{cls}(\boldsymbol{C})"']
\end{tikzcd}$$

**Figure 9: Mediating 2-cell**

```
(15) (KIF$function mediator)
     (= (KIF$source mediator) concept)
     (= (KIF$target mediator) 2-cell)
     (forall (?c (concept ?c))
         (and (= (source (mediator ?c)) ?c)
              (= (target (mediator ?c)) (terminal (classification ?c)))
              (= (index-function (mediator ?c)) (mediator-function ?c))))
```

○ For any collective concept $\boldsymbol{C}$ and any class function $f : A \rightarrow \mathsf{index}(\boldsymbol{C})$ whose target is the index class of $\boldsymbol{C}$, the quadruple $f^{-1}(\boldsymbol{C}) = \langle \mathsf{cls}(\boldsymbol{C}), \mathsf{src}(f), \mathsf{ext}(\boldsymbol{C}) \circ f^{\mathrm{op}}, f \circ \mathsf{int}(\boldsymbol{C})\rangle$ is also a collective concept, and the function $f$ is the index function of a 2-cell with source $f^{-1}(\boldsymbol{C})$ and target $\boldsymbol{C}$. The closure conditions for the inverse image concept follow from basic properties of functions and residuation. Hence, for a fixed classification $\boldsymbol{A}$ the function $\mathsf{index} : \mathsf{CLASSIFICATION} \rightarrow \mathsf{Class}$ is part of a fibration.

```
(16) (KIF$opspan inverse-image-opspan)
     (= 2-cell-opspan [SET.FTN$target CLS$concept])

(17) (KIF$relation invertible)
     (= (KIF$collection1 invertible) SET.FTN$function)
     (= (KIF$collection2 invertible) CLS$classification)
     (= (KIF$extent invertible) (KIF$pullback inverse-image-opspan))

(18) (KIF$function inverse-image)
     (= (KIF$source inverse-image) (KIF$pullback inverse-image-opspan))
     (= (KIF$target inverse-image) concept)
     (forall (?f (SET.FTN$function ?f)
              ?a (CLS$classification ?a) (invertible ?f ?a))
         (and (= (classification (inverse-image [?f ?a])) ?a)
              (= (index (inverse-image [?f ?a])) (SET.FTN$source ?f))
              (= (extent (inverse-image [?f ?a]))
                 (REL$composition [(extent ?a) (REL$opposite (SET.FTN$fn2rel ?f))]))
              (= (intent (inverse-image [?f ?a]))
                 (REL$composition [?f (intent ?a)]))))

(19) (KIF$function inverse-image-mediator)
     (= (KIF$source inverse-image-mediator) (KIF$pullback inverse-image-opspan))
     (= (KIF$target inverse-image-mediator) 2-cell)
     (forall (?f (SET.FTN$function ?f)
              ?a (CLS$classification ?a) (invertible ?f ?a))
         (and (= (source (inverse-image-mediator [?f ?a])) (inverse-image [?f ?a]))
              (= (target (inverse-image-mediator [?f ?a])) ?a)
              (= (index-function (inverse-image-mediator [?f ?a])) ?f)))
```

○ A collective concept $\langle \boldsymbol{A}, A, X, Y\rangle = \langle \mathit{cls}(\boldsymbol{C}), \mathit{ind}(\boldsymbol{C}), \mathit{ext}(\boldsymbol{C}), \mathit{int}(\boldsymbol{C})\rangle$ is also called a *concept space*. The indexed or named formal concepts in a concept space are also called *conceptual views*. Conceptual knowledge is represented by the following components of a concept space.
- The three preorders on instances, types and conceptual views.
- The membership or instantiation relation between instances and conceptual views, whose columns, when regarded as a Boolean matrix, record the *extent* of all the conceptual views.

$$\begin{tikzcd}
 & & \mathit{typ}(\boldsymbol{A}) \arrow[dl, "\tau_A"] \\
A \arrow[urr, "Y"] \arrow[r, "\mu"] & L(\boldsymbol{A}) & \\
 & & \mathit{inst}(\boldsymbol{A}) \arrow[ul, "\iota_A"'] \arrow[ull, "X"] \arrow[uu, "\boldsymbol{A}"']
\end{tikzcd}$$

| | |
|---|---|
| $\boldsymbol{A}$ | *classification* |
| $\mathit{inst}(\boldsymbol{A})$ | *instance preorder* |
| $\mathit{typ}(\boldsymbol{A})$ | *type preorder* |
| $A$ | *index preorder* |
| $X$ | *extent* |
| $Y$ | *intent* |
| $L(\boldsymbol{A})$ | *concept lattice* |

- The abstraction relation between conceptual views and types, whose rows, when regarded as a Boolean matrix, record the *intent* of all the conceptual views.
- The *classification* relation between instances and types.

The constraining relationships between the extent and intent of a concept space can be expressed in three different forms: residuation, inclusion and derivation.

| incidence constraints | | |
|---|---|---|
| residuation | inclusion | derivation |
| $Y = X\backslash\boldsymbol{A}$<br>$X = \boldsymbol{A}/Y$ | $\forall_{a \in A,\, t \in \mathit{typ}(\boldsymbol{A})}\, aYt$ *iff* $Xa \subseteq \boldsymbol{A}t$<br>$\forall_{i \in \mathit{inst}(\boldsymbol{A}),\, a \in A}\, iXa$ *iff* $i\boldsymbol{A} \supseteq aY$ | $\forall_{a \in A}\ aY = (Xa)'$<br>$\forall_{a \in A}\ Xa = (aY)'$ |

In order to construct a concept space, we start out by specifying the class $A$ to be a collection of names, with each $a \in A$ representing a subset of types $aY_0 \subseteq \mathit{typ}(\boldsymbol{A})$. We use the two residuation operators, which are a generalized form of the derivation operators of Formal Concept Analysis, in order to define the notion of a *collective formal concept*. We define the extent $X$ to be the right residuation of $\boldsymbol{A}$ along $Y_0$:

$$X = \boldsymbol{A}/Y_0 = \{(i, a) \mid \forall_{t \in \mathit{typ}(\boldsymbol{A})}\, (aY_0t \Rightarrow i\boldsymbol{A}t)\}.$$

so that $X \circ Y_0 \subseteq \boldsymbol{A}$. We define $Y$ to be the left residuation of $\boldsymbol{A}$ along $X$:

$$Y = X\backslash\boldsymbol{A} = \{(a, t) \mid \forall_{i \in \mathit{inst}(\boldsymbol{A})}\, (iXa \Rightarrow i\boldsymbol{A}t)\}.$$

From these definitions it is straightforward to show that $X$ is the right residuation, $X = \boldsymbol{A}/Y$, of $\boldsymbol{A}$ along $Y$: First of all, since $Y_0 \subseteq X\backslash\boldsymbol{A} = Y$, by contravariance of residuation $X = \boldsymbol{A}/Y_0 \supseteq \boldsymbol{A}/Y$; and secondly, since $X \circ Y = X \circ (X\backslash\boldsymbol{A}) \subseteq \boldsymbol{A}$, we have $X \subseteq \boldsymbol{A}/Y$.

**Table 3: KIF Functions used to define the Concept Lattice Functor**

| |
|---|
| left derivation : Classification → Function |
| right derivation : Classification → Function |
| instance closure : Classification → Function |
| type closure : Classification → Function |
| concept : Classification → Class |
| extent : Classification → Function |
| intent : Classification → Function |
| instance concept : Classification → Function |
| type concept : Classification → Function |
| concept order : Classification → Partial Order |
| meet : Classification → Function |
| join : Classification → Function |
| complete lattice : Classification → Complete Lattice |
| adjoint pair : Infomorphism → Adjoint Pair |
| instance embedding : Classification → Function |
| type embedding : Classification → Function |
| **concept lattice : Classification → Concept Lattice** |
| **concept morphism : Infomorphism → Concept Morphism** |
| instance embedding relation : Classification → Relation |
| type embedding relation : Classification → Relation |

**Table 4: Functors and Natural Isomorphisms**

| | |
|---|---|
| $L \circ C = Id_{\mathrm{CLASSIFICATION}}$ | `CLS.CL$concept-lattice . CL$classification = Id` |
| | `CLS.INFO$concept-morphism . CL.MOR$infomorphism = Id` |
| $C \circ L \cong Id_{\mathrm{CLASSIFICATION}}$ | `CL$classification . CLS.CL$concept-lattice ≅ Id` |
| | `CL.MOR$infomorphism . CLS.INFO$concept-morphism ≅ Id` |

## *Functional Infomorphisms*

**`CLS.INFO`**

o Classifications are related through (functional) infomorphisms. A (*functional*) *infomorphism* $\boldsymbol{f} : \boldsymbol{A} \rightleftarrows \boldsymbol{B}$ from classification $\boldsymbol{A}$ to classification $\boldsymbol{B}$ (Figure 10) consists of a pair $\boldsymbol{f} = \langle inst(\boldsymbol{f}), typ(\boldsymbol{f})\rangle$ of oppositely directed functions,

– a function between instances $inst(\boldsymbol{f}) : inst(\boldsymbol{B}) \rightarrow inst(\boldsymbol{A})$ and

– a function between types $typ(\boldsymbol{f}) : typ(\boldsymbol{A}) \rightarrow typ(\boldsymbol{B})$,

which satisfy the fundamental property:

$$inst(\boldsymbol{f})(i) \vDash_{\boldsymbol{A}} t \text{ iff } i \vDash_{\boldsymbol{B}} typ(\boldsymbol{f})(t)$$

$$\begin{tikzcd}
typ(\boldsymbol{A}) \arrow[r, "typ(\boldsymbol{f})"] & typ(\boldsymbol{B}) \\
inst(\boldsymbol{A}) \arrow[u, "\boldsymbol{A}"'] & inst(\boldsymbol{B}) \arrow[l, "inst(\boldsymbol{f})"] \arrow[u, "\boldsymbol{B}"']
\end{tikzcd}$$

**Figure 10: Functional Infomorphism**

for all instances $i \in inst(\boldsymbol{B})$ and all types $t \in typ(\boldsymbol{A})$. The relation expressed on the either side of this equivalence is the bond of the relational infomorphism generated by this functional infomorphism. This *relational infomorphism* is defined below in terms of the left relation of the *instance monotonic function* and the right relation of the *type monotonic function*.

The following is a KIF representation for the elements of an infomorphism. Such elements are useful for the definition of an infomorphism. In the same fashion as classifications, infomorphisms are specified by declaration and population. The term *infomorphism* allows one to *declare* infomorphisms themselves, and the two terms *source* and *target* allow one to *declare* their associated source (domain) and target (codomain) classifications, respectively. The terms *instance* and *type* resolve infomorphisms into their parts, thus allowing one to *populate* infomorphisms. Let CLASSIFICATION denote the quasi-category of classifications and infomorphisms.

```
(1) (KIF$collection infomorphism)

(2) (KIF$function source)
    (= (KIF$source source) infomorphism)
    (= (KIF$target source) CLS$classification)
```

```
(3) (KIF$function target)
    (= (KIF$source target) infomorphism)
    (= (KIF$target target) CLS$classification)

(4) (KIF$function instance)
    (= (KIF$source instance) infomorphism)
    (= (KIF$target instance) SET.FTN$function)
    (forall (?f (infomorphism ?f))
        (and (= (SET.FTN$source (instance ?f)) (CLS$instance (target ?f)))
             (= (SET.FTN$target (instance ?f)) (CLS$instance (source ?f)))))

(5) (KIF$function type)
    (= (KIF$source type) infomorphism)
    (= (KIF$target type) SET.FTN$function)
    (forall (?f (infomorphism ?f))
        (and (= (SET.FTN$source (type ?f)) (CLS$type (source ?f)))
             (= (SET.FTN$target (type ?f)) (CLS$type (target ?f)))))

(6) (forall (?f (infomorphism ?f)
             ?i ((CLS$instance (target ?f)) ?i)
             ?t ((CLS$type (source ?f)) ?t))
        (<=> ((source ?f) ((instance ?f) ?i) ?t)
             ((target ?f) ?i ((type ?f) ?t))))
```

- The fundamental property of an infomorphism $\boldsymbol{f} : \boldsymbol{A} \rightleftarrows \boldsymbol{B}$ expresses the bonding classification of *bond*($\boldsymbol{f}$) : $\boldsymbol{A} \rightleftarrows \boldsymbol{B}$, the bond associated with the infomorphism. This can be equivalently define in terms of either the instance or the type function. Here we use the instance function. For comparison, see the right operator that maps a function to a relation in the presence of a preorder.

```
(7) (KIF$function bond)
    (= (KIF$source bond) infomorphism)
    (= (KIF$target bond) CLS.BND$bond)
    (forall (?f (infomorphism ?f))
        (and (= (CLS.BND$source (bond ?f)) (source ?f))
             (= (CLS.BND$target (bond ?f)) (target ?f))
             (forall (?i ((CLS$instance (target ?f)) ?i)
                      ?t ((CLS$type (source ?f)) ?t))
             (<=> ((CLS.BND$classification (bond ?f)) ?i ?t)
                  ((source ?f) ((instance ?f) ?i) ?t))))
```

- The *instance monotonic function* is the instance function regarded as a monotonic function between instance orders. Dually, the *type monotonic function* is the type function regarded as a monotonic function between type orders.

```
(8) (KIF$function monotonic-instance)
    (= (KIF$source monotonic-instance) infomorphism)
    (= (KIF$target monotonic-instance) ORD.FTN$monotonic-function)
    (forall (?f (infomorphism ?f))
        (and (= (ORD.FTN$source (monotonic-instance ?f))
                (CLS.CL$instance-order (target ?f)))
             (= (ORD.FTN$target (monotonic-instance ?f))
                (CLS.CL$instance-order (source ?f)))
             (= (ORD.FTN$function (monotonic-instance ?f)) (instance ?f))))

(9) (KIF$function monotonic-type)
    (= (KIF$source monotonic-type) infomorphism)
    (= (KIF$target monotonic-type) ORD.FTN$monotonic-function)
    (forall (?f (infomorphism ?f))
        (and (= (ORD.FTN$source (monotonic-type ?f)) (CLS.CL$type-order (source ?f)))
             (= (ORD.FTN$target (monotonic-type ?f)) (CLS.CL$type-order (target ?f)))
             (= (ORD.FTN$function (monotonic-type ?f)) (type ?f))))
```

- The *instance relation* is the left relation of the instance monotonic function. The *type relation* is the right relation of the type monotonic function.

```
(10) (KIF$function relational-instance)
     (= (KIF$source relational-instance) infomorphism)
     (= (KIF$target relational-instance) REL$relation)
```

```
        (forall (?f (infomorphism ?f))
            (and (= (REL$source (relational-instance ?f)) (instance (source ?f)))
                 (= (REL$target (relational-instance ?f)) (instance (target ?f)))
                 (= (relational-instance ?f)
                    (SET.FTN$left [(instance ?f) (source ?f)]))))

(11) (KIF$function relational-type)
     (= (KIF$source relational-type) infomorphism)
     (= (KIF$target relational-type) REL$relation)
     (forall (?f (infomorphism ?f))
         (and (= (REL$source (relational-type ?f)) (type (source ?f)))
              (= (REL$target (relational-type ?f)) (type (target ?f)))
              (= (relational-type ?f)
                 (SET.FTN$right [(type ?f) (target ?f)]))))
```

o Any functional infomorphism can be transformed into a *relational infomorphism*.

```
(12) (KIF$function relational-infomorphism)
     (KIF$function fn2rel)
     (= fn2rel relational-infomorphism)
     (= (KIF$source relational-infomorphism) infomorphism)
     (= (KIF$target relational-infomorphism) CLS.REL$infomorphism)
     (forall (?f (infomorphism ?f))
         (and (= (CLS.REL$source (relational-infomorphism ?f)) (source ?f))
              (= (CLS.REL$target (relational-infomorphism ?f)) (target ?f))
              (= (CLS.REL$instance (relational-infomorphism ?f))
                 (relational-instance ?f))
              (= (CLS.REL$type (relational-infomorphism ?f))
                 (relational-type ?f))))
```

o It can be shown that the bond of the relational infomorphism of a functional infomorphism ***f*** is the bond associated with ***f***.

```
     (forall (?f (infomorphism ?f))
         (= (CLS.REL$bond (relational-infomorphism ?f)) (bond ?f)))
```

o The *composition* function operates on any two infomorphisms that are *composable* in the sense that the target classification of the first is equal to the source classification of the second. Composition produces an infomorphism, whose instance function is the composition of the instance functions of the components and whose type function is the composition of the type functions of the components.

```
(13) (KIF$opspan composable-opspan)
     (= composable-opspan [target source])

(14) (KIF$relation composable)
     (= (KIF$collection1 composable) infomorphism)
     (= (KIF$collection2 composable) infomorphism)
     (= (KIF$extent composable) (KIF$pullback composable-opspan))

(15) (KIF$function composition)
     (= (KIF$source composition) (KIF$pullback composable-opspan))
     (= (KIF$target composition) infomorphism)
     (forall (?f1 (infomorphism ?f1) ?f2 (infomorphism ?f2) (composable ?f1 ?f2))
         (and (= (source (composition [?f1 ?f2])) (source ?f1))
              (= (target (composition [?f1 ?f2])) (target ?f2))
              (= (instance (composition [?f1 ?f2]))
                 (SET.FTN$composition [(instance ?f2) (instance ?f1)]))
              (= (type (composition [?f1 ?f2]))
                 (SET.FTN$composition [(type ?f) (type ?g)]))))
```

o The *identity* function associates a well-defined identity infomorphism with any classification, whose instance function is the identity class function on instances and whose type function is the identity class function on types.

```
(16) (KIF$function identity)
     (= (KIF$source identity) CLS$classification)
     (= (KIF$target identity) infomorphism)
     (forall (?c (CLS$classification ?c))
         (and (= (source (identity ?c)) ?c)
              (= (target (identity ?c)) ?c)
```

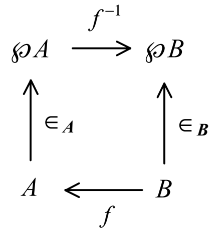
```
                    (= (instance (identity ?c))
                       (SET.FTN$identity (CLS$instance ?c)))
                    (= (type (identity ?c))
                       (SET.FTN$identity (CLS$type ?c)))))
```

- Duality can be extended to infomorphisms. For any infomorphism $\boldsymbol{f} : \boldsymbol{A} \rightleftarrows \boldsymbol{B}$, the *opposite* or *dual* of $\boldsymbol{f}$ is the infomorphism $\boldsymbol{f}^{\perp} : \boldsymbol{B}^{\perp} \rightleftarrows \boldsymbol{A}^{\perp}$ whose source classification is the opposite of the target classification of $\boldsymbol{f}$, whose target classification is the opposite of the source classification of $\boldsymbol{f}$, whose instance function is the type function of $\boldsymbol{f}$, whose type function is the instance function of $\boldsymbol{f}$, and whose fundamental condition is equivalent to that of $\boldsymbol{f}$:

  $typ(f)(t) \models^{\perp} i$ iff $t \models^{\perp} inst(f)(i)$.

```
(17) (KIF$function opposite)
     (= (KIF$source opposite) infomorphism)
     (= (KIF$target opposite) infomorphism)
     (forall (?f (infomorphism ?f))
         (and (= (source (opposite ?f)) (CLS$opposite (target ?f)))
              (= (target (opposite ?f)) (CLS$opposite (source ?f)))
              (= (instance (opposite ?f)) (type ?f))
              (= (type (opposite ?f)) (instance ?f))))
```

- For any class function $f : B \rightarrow A$ the components of the *instance power infomorphism* $\wp f : \wp A \rightleftarrows \wp B$ over $f$ (Figure 11) is defined as follows: the source classification $\wp A = \langle A, \wp A, \in_A \rangle$ is the instance power classification over the target class $A$, the target classification $\wp B = \langle B, \wp B, \in_B \rangle$ is the instance power classification over the source class $B$, the instance function is $f$, and the type function is the inverse image function $f^{-1} : \wp A \rightarrow \wp B$ from the power-class of $A$ to the power-class of $B$. Note the contravariance.

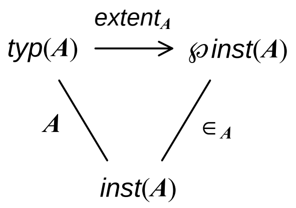


$$\begin{tikzcd} \wp A \arrow[r, "f^{-1}"] & \wp B \\ A \arrow[u, "\in_A"'] & B \arrow[l, "f"] \arrow[u, "\in_B"'] \end{tikzcd}$$

**Figure 11: Instance Power Infomorphism**

```
(18) (KIF$function instance-power)
     (= (KIF$source instance-power) SET.FTN$function)
     (= (KIF$target instance-power) infomorphism)
     (forall (?f (SET.FTN$function ?f))
         (and (= (source (instance-power ?f))
                 (SET.FTN$power (SET.FTN$target ?f)))
              (= (target (instance-power ?f))
                 (SET.FTN$power (SET.FTN$source ?f)))
              (= (instance (instance-power ?f)) ?f)
              (= (type (instance-power ?f))
                 (SET.FTN$inverse-image ?f))))
```

- It is a standard fact in Information Flow that from any classification $\boldsymbol{A}$ there is a *canonical extent infomorphism* $\eta_{\boldsymbol{A}} : \boldsymbol{A} \rightleftarrows \wp inst(\boldsymbol{A})$ (Figure 12) from $\boldsymbol{A}$ to the instance power classification $\wp inst(\boldsymbol{A}) = \langle inst(\boldsymbol{A}), \wp inst(\boldsymbol{A}), \in \rangle$ over the instance class. This infomorphism is the $\boldsymbol{A}$th component of a natural quasi-transformation called *eta*, the unit of the adjunction between the underlying instance functor and the instance power functor. The instance function of eta is the identity function on the instance class $inst(\boldsymbol{A})$, and the type function of eta is the extent function $extent_{\boldsymbol{A}} : typ(\boldsymbol{A}) \rightarrow \wp inst(\boldsymbol{A})$.



$$\begin{tikzcd} typ(A) \arrow[rr, "extent_A"] \arrow[dr, dash, "A"'] & & \wp inst(A) \arrow[dl, dash, "\in_A"] \\ & inst(A) & \end{tikzcd}$$

**Figure 12: Extent Infomorphism $\eta_A$**

```
(19) (KIF$function eta)
     (= (KIF$source eta) CLS$classification)
     (= (KIF$target eta) infomorphism)
     (forall (?a (classification ?a))
         (and (= (source (eta ?a)) ?a)
              (= (target (eta ?a)) (CLS$instance-power (CLS$instance ?a)))
              (= (instance (eta ?a)) (SET.FTN$identity (CLS$instance ?a)))
              (= (type (eta ?a)) (CLS$extent ?a))))
```

- Let $\boldsymbol{f} = \langle inst(\boldsymbol{f}), typ(\boldsymbol{f}) \rangle : \boldsymbol{A} \rightleftarrows \boldsymbol{B}$ be any infomorphism from classification $\boldsymbol{A}$ to classification $\boldsymbol{B}$ with instance function $inst(\boldsymbol{f}) : inst(\boldsymbol{B}) \rightarrow inst(\boldsymbol{A})$ and type function $typ(\boldsymbol{f}) : typ(\boldsymbol{A}) \rightarrow typ(\boldsymbol{B})$. There is an adjoint pair $adj(\boldsymbol{f}) = \langle left(adj(\boldsymbol{f})), right(adj(\boldsymbol{f}))\rangle : complete\text{-}lattice(\boldsymbol{A}) \rightleftarrows complete\text{-}lattice(\boldsymbol{B})$, defined as follows.

$$\begin{aligned} &left(adj(\boldsymbol{f}))(d) \\ &\quad = left(adj(\boldsymbol{f}))(\langle extent_{\boldsymbol{B}}(d), intent_{\boldsymbol{B}}(d)\rangle) \\ &\quad = \langle (typ(\boldsymbol{f})^{-1}(intent_{\boldsymbol{B}}(d)))',\ (typ(\boldsymbol{f})^{-1}(intent_{\boldsymbol{B}}(d))\rangle \end{aligned}$$

for all concepts $d \in complete\text{-}lattice(\boldsymbol{B})$, and

$$\begin{aligned} &right(adj(\boldsymbol{f}))(c) \\ &\quad = right(adj(\boldsymbol{f}))(\langle extent_{\boldsymbol{A}}(c), intent_{\boldsymbol{A}}(c)\rangle) \\ &\quad = \langle (inst(\boldsymbol{f})^{-1}(extent_{\boldsymbol{A}}(c)), (inst(\boldsymbol{f})^{-1}(extent_{\boldsymbol{A}}(c)))'\rangle \end{aligned}$$

for all concepts $c \in complete\text{-}lattice(\boldsymbol{A})$.

```
(20) (KIF$function adjoint-pair)
     (= (KIF$source adjoint-pair) infomorphism)
     (= (KIF$target adjoint-pair) LAT.ADJ$adjoint-pair)
     (forall (?f (infomorphism ?f))
         (and (= (LAT.ADJ$source (adjoint-pair ?f))
                 (CLS.CL$complete-lattice (target ?f)))
              (= (LAT.ADJ$target (adjoint-pair ?f))
                 (CLS.CL$complete-lattice (source ?f)))
              (= (SET.FTN$composition
                     [(ORD.FTN$function (LAT.ADJ$left (adjoint-pair ?f)))
                      (CLS.CL$intent (source ?f))])
                 (SET.FTN$composition
                     [(CLS.CL$intent (target ?f))
                      (SET.FTN$inverse-image (type ?f))]))
              (= (SET.FTN$composition
                     [(ORD.FTN$function (LAT.ADJ$right (adjoint-pair ?f)))
                      (CLS.CL$extent (target ?f))])
                 (SET.FTN$composition
                     [(CLS.CL$extent (source ?f))
                      (SET.FTN$inverse-image (instance ?f))]))))
```

o Let $\boldsymbol{f} : \boldsymbol{A} \rightleftarrows \boldsymbol{B}$ be any infomorphism from classification $\boldsymbol{A}$ to classification $\boldsymbol{B}$ with instance function $inst(\boldsymbol{f}) : inst(\boldsymbol{B}) \rightarrow inst(\boldsymbol{A})$ and type function $typ(\boldsymbol{f}) : typ(\boldsymbol{A}) \rightarrow typ(\boldsymbol{B})$. There is a concept morphism

$$concept\text{-}morphism(\boldsymbol{f}) = \langle inst(\boldsymbol{A}), typ(\boldsymbol{A}), adj(\boldsymbol{A})\rangle : concept\text{-}lattice(\boldsymbol{A}) \rightleftarrows concept\text{-}lattice(\boldsymbol{B}),$$

whose instance/type functions are the same as $\boldsymbol{f}$, and whose adjoint pair the adjoint pair of $\boldsymbol{f}$.

The concept morphism associated with an infomorphism is the image of the morphism function of the concept lattice functor applied to the infomorphism:

$$L : \text{CLASSIFICATION} \rightarrow \text{CONCEPT LATTICE}.$$

```
(21) (KIF$function concept-morphism)
     (= (KIF$source concept-morphism) infomorphism)
     (= (KIF$target concept-morphism) CL.MOR$concept-morphism)
     (forall (?f (infomorphism ?f))
         (and (= (CL.MOR$source (concept-morphism ?f))
                 (CLS.CL$concept-lattice (source ?f)))
              (= (CL.MOR$target (concept-morphism ?f))
                 (CLS.CL$concept-lattice (target ?f)))
              (= (CL.MOR$adjoint-pair (concept-morphism ?f)) (adjoint-pair ?f))
              (= (CL.MOR$instance (concept-morphism ?f)) (instance ?f))
              (= (CL.MOR$type (concept-morphism ?f)) (type ?f))))
```

## *Relational Infomorphisms*

**CLS.REL**

$$\begin{tikzcd}
\mathit{typ}(\boldsymbol{A}) \arrow[r, "\mathit{typ}(\boldsymbol{r})"] & \mathit{typ}(\boldsymbol{B}) \\
\mathit{inst}(\boldsymbol{A}) \arrow[u, "\boldsymbol{A}"'] \arrow[r, "\mathit{inst}(\boldsymbol{r})"'] & \mathit{inst}(\boldsymbol{B}) \arrow[u, "\boldsymbol{B}"'] \arrow[ul, "\mathit{bond}(\boldsymbol{f})"']
\end{tikzcd}$$

**Figure 13: Relational Infomorphism**

- Classifications are also related through (relational) infomorphisms. A (*relational*) *infomorphism* $\boldsymbol{r} : \boldsymbol{A} \rightrightarrows \boldsymbol{B}$ from classification $\boldsymbol{A}$ to classification $\boldsymbol{B}$ (Figure 13) is a pair $\boldsymbol{r} = \langle \mathit{inst}(\boldsymbol{r}), \mathit{typ}(\boldsymbol{r}) \rangle$ of binary relations in the same direction,
  - a relation between instances $\mathit{inst}(\boldsymbol{r}) : \mathit{inst}(\boldsymbol{A}) \rightarrow \mathit{inst}(\boldsymbol{B})$ and
  - a relation between types $\mathit{typ}(\boldsymbol{r}) : \mathit{typ}(\boldsymbol{A}) \rightarrow \mathit{typ}(\boldsymbol{B})$,

  which satisfy the fundamental property:

  $$\mathit{inst}(\boldsymbol{r}) \backslash \boldsymbol{A} = \boldsymbol{B} / \mathit{typ}(\boldsymbol{r}).$$

```
(1) (KIF$collection infomorphism)

(2) (KIF$function source)
    (= (KIF$source source) infomorphism)
    (= (KIF$target source) CLS$classification)

(3) (KIF$function target)
    (= (KIF$source target) infomorphism)
    (= (KIF$target target) CLS$classification)

(4) (KIF$function instance)
    (= (KIF$source instance) infomorphism)
    (= (KIF$target instance) REL$relation)
    (forall (?r (infomorphism ?r))
        (and (= (REL$source (instance ?r)) (CLS$instance (source ?r)))
             (= (REL$target (instance ?r)) (CLS$instance (target ?r)))))

(5) (KIF$function type)
    (= (KIF$source type) infomorphism)
    (= (KIF$target type) REL$relation)
    (forall (?r (infomorphism ?r))
        (and (= (REL$source (type ?r)) (CLS$type (source ?r)))
             (= (REL$target (type ?r)) (CLS$type (target ?r)))))

(6) (forall (?r (infomorphism ?r))
        (= (REL$left-residuation [(instance ?r) (source ?r)])
           (REL$right-residuation [(type ?r) (target ?r)])))
```

- For any relational infomorphism $\boldsymbol{r} : \boldsymbol{A} \rightrightarrows \boldsymbol{B}$, the common relation $\mathit{inst}(\boldsymbol{r}) \backslash \boldsymbol{A} = \boldsymbol{B} / \mathit{typ}(\boldsymbol{r})$ in the fundamental property, consider to be a classification $\mathit{bond}(\boldsymbol{r}) = \langle \mathit{inst}(\boldsymbol{B}), \mathit{typ}(\boldsymbol{A}), \models_{\mathit{bond}(r)} \rangle$, is called the *bond* of $\boldsymbol{r}$ – it bonds the instance and type classifications into a unity.

```
(7) (KIF$function bond)
    (= (KIF$source bond) infomorphism)
    (= (KIF$target bond) CLS.BND$bond)
    (forall (?r (infomorphism ?r))
        (and (= (CLS.BND$source (bond ?r)) (source ?r))
             (= (CLS.BND$target (bond ?r)) (target ?r))
             (= (CLS.BND$classification (bond ?r))
                (REL$left-residuation [(instance ?r) (source ?r)]))))
```

- Given any two relational infomorphisms $\langle r_1, s_1 \rangle : \boldsymbol{A} \rightrightarrows \boldsymbol{B}$ and $\langle r_2, s_2 \rangle : \boldsymbol{B} \rightrightarrows \boldsymbol{C}$, which are *composable* in the sense the target classification of the first is the source classification of the second, there is a *composite* infomorphism $\langle r_1, s_1 \rangle \circ \langle r_2, s_2 \rangle = \langle r_1 \circ r_2, s_1 \circ s_2 \rangle : \boldsymbol{A} \rightrightarrows \boldsymbol{C}$ defined by composing the type and instance relations – its fundamental property follows from composition and associative laws.

```
(8) (KIF$opspan composable-opspan)
    (= composable-opspan [target source])

(9) (KIF$relation composable)
    (= (KIF$collection1 composable) infomorphism)
```

```
        (= (KIF$collection2 composable) infomorphism)
        (= (KIF$extent composable) (KIF$pullback composable-opspan))

    (10) (KIF$function composition)
         (= (KIF$source composition) (KIF$pullback composable-opspan))
         (= (KIF$target composition) infomorphism)
         (forall (?r1 (infomorphism ?r1) ?r2 (infomorphism ?r2) (composable ?r1 ?r2))
             (and (= (source (composition [?r1 ?r2])) (source ?r1))
                  (= (target (composition [?r1 ?r2])) (target ?r2))
                  (= (instance (composition [?r1 ?r2]))
                     (REL$composition [(instance ?r1) (instance ?r2)]))
                  (= (type (composition [?r1 ?r2]))
                     (REL$composition [(type ?r1) (type ?r2)]))))
```

- Given any classification $\boldsymbol{A} = \langle inst(\boldsymbol{A}), typ(\boldsymbol{A}), \models_{\boldsymbol{A}} \rangle$, the pair of identity relations on types and instances, with the bond being $\boldsymbol{A}$, forms an *identity* infomorphism $Id_{\boldsymbol{A}} : \boldsymbol{A} \rightrightarrows \boldsymbol{A}$ (with respect to composition).

```
    (11) (KIF$function identity)
         (= (KIF$source identity) CLS$classification)
         (= (KIF$target identity) infomorphism)
         (forall (?a (CLS$classification ?a))
             (and (= (source (identity ?a)) ?a)
                  (= (target (identity ?a)) ?a)
                  (= (instance (identity ?a)) (REL$identity (CLS$instance ?a)))
                  (= (type (identity ?a)) (REL$identity (CLS$type ?a)))))
```

- For any given infomorphism $\langle r, s \rangle : \boldsymbol{A} \rightrightarrows \boldsymbol{B}$ the *dual* infomorphism $\langle r, s \rangle^{\mathrm{op}} = \langle s^{\mathrm{op}}, r^{\mathrm{op}} \rangle : \boldsymbol{B}^{\mathrm{op}} \rightrightarrows \boldsymbol{A}^{\mathrm{op}}$ is the relational infomorphism with type and instance relations switched and transposed.

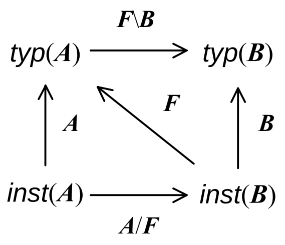
```
    (12) (KIF$function opposite)
         (= (KIF$source opposite) infomorphism)
         (= (KIF$target opposite) infomorphism)
         (forall (?r (infomorphism ?r))
             (and (= (source (opposite ?r)) (CLS$opposite (target ?r)))
                  (= (target (opposite ?r)) (CLS$opposite (source ?r)))
                  (= (instance (opposite ?r)) (REL$opposite (type ?r)))
                  (= (type (opposite ?r)) (REL$opposite (instance ?r)))))
```

- The fundamental property of relational infomorphisms for composition, identity and involution follow from basic properties of residuation.

## Bonds

**CLS.BND**

- Classifications are also related through bonds. A *bond* $\boldsymbol{F} : \boldsymbol{A} \rightharpoondown \boldsymbol{B}$ from classification $\boldsymbol{A}$ to classification $\boldsymbol{B}$ (Figure 14) is a classification $cls(\boldsymbol{F}) = \langle inst(\boldsymbol{B}), typ(\boldsymbol{A}), \models_{\boldsymbol{F}} \rangle$ sharing types with $\boldsymbol{A}$ and instances with $\boldsymbol{B}$, that is compatible with $\boldsymbol{A}$ and $\boldsymbol{B}$ in the sense of closure: type classes $\{i\boldsymbol{F} \mid i \in inst(\boldsymbol{B})\}$ are intents of $\boldsymbol{A}$ and instance classes $\{\boldsymbol{F}t \mid t \in typ(\boldsymbol{B})\}$ are extents of $\boldsymbol{B}$. Closure can be expressed relationally (in terms of residuation) as the following fundamental closure properties.



$$\begin{tikzcd} typ(\boldsymbol{A}) \arrow[r, "\boldsymbol{F}\backslash\boldsymbol{B}"] & typ(\boldsymbol{B}) \\ inst(\boldsymbol{A}) \arrow[u, "\boldsymbol{A}"] \arrow[r, "\boldsymbol{A}/\boldsymbol{F}"'] & inst(\boldsymbol{B}) \arrow[u, "\boldsymbol{B}"'] \arrow[ul, "\boldsymbol{F}"'] \end{tikzcd}$$

**Figure 14: Bond**

$(\boldsymbol{A}/\boldsymbol{F})\backslash\boldsymbol{A} = \boldsymbol{F}$ and $\boldsymbol{B}/(\boldsymbol{F}\backslash\boldsymbol{B}) = \boldsymbol{F}$.

The first expression says that $\langle \boldsymbol{A}/\boldsymbol{F}, \boldsymbol{F} \rangle$ is an $inst(\boldsymbol{B})$-indexed collective $\boldsymbol{A}$-concept (or $\boldsymbol{F}$ is a collective $\boldsymbol{A}$-intent), and the second expression says that $\langle \boldsymbol{F}, \boldsymbol{F}\backslash\boldsymbol{B} \rangle$ is an $typ(\boldsymbol{A})$-indexed collective $\boldsymbol{B}$-concept (or that $\boldsymbol{F}$ is a collective $\boldsymbol{B}$-extent). The first expression also says that classification $\boldsymbol{F}$ is type-closed with respect to classification $\boldsymbol{A}$, and the second expression says that classification $\boldsymbol{F}$ is instance-closed with respect to classification $\boldsymbol{B}$. Let BOND denote the quasi-category of classifications and bonds.

```
    (1) (KIF$collection bond)

    (2) (KIF$function source)
        (= (KIF$source source) bond)
        (= (KIF$target source) CLS$classification)
```

```
(3) (KIF$function target)
    (= (KIF$source target) bond)
    (= (KIF$target target) CLS$classification)

(4) (KIF$function classification)
    (= (KIF$source classification) bond)
    (= (KIF$target classification) CLS$classification)
    (forall (?b (bond ?b))
        (and (= (CLS$instance (classification ?b)) (CLS$instance (target ?b)))
             (= (CLS$type (classification ?b)) (CLS$type (source ?b)))))

(5) (forall (?b (bond ?b))
        (and (= (REL$left-residuation
                    [(REL$right-residuation [(classification ?b) (source ?b)])
                     (source ?b)])
                (classification ?b))
             (= (REL$right-residuation
                    [(REL$left-residuation [(classification ?b) (target ?b)])
                     (target ?b)])
                (classification ?b))))
```

- A bond has an associated *bimodule*, since the classification of a bond, as a relation, is order-closed on left and right:

  $j' \leq_B j$, $jFt$ imply $j'Ft$, and $jFt$, $t \leq_A t'$ imply $jFt'$; or,

  $j' \leq_B j$ implies $j'F \supseteq jF$, and $t \leq_A t'$ implies $Ft \subseteq Ft'$.

```
(6) (KIF$function bimodule)
    (= (KIF$source bimodule) bond)
    (= (KIF$target bimodule) ORD$bimodule)
    (forall (?b (bond ?b))
        (and (= (ORD.REL$source (bimodule ?b)) (CLS.CL$instance-order (target ?b)))
             (= (ORD.REL$target (bimodule ?b)) (CLS.CL$type-order (source ?b)))
             (= (ORD.REL$relation (bimodule ?b)) (classification ?b))))
```

- For any bond $\boldsymbol{F} : \boldsymbol{A} \rightharpoonup \boldsymbol{B}$, the residuations in the fundamental closure property form a *relation infomorphism* $\mathsf{info}(\boldsymbol{F}) = \langle(\boldsymbol{A}/\boldsymbol{F}), (\boldsymbol{F}\backslash\boldsymbol{B})\rangle : \boldsymbol{A} \rightleftarrows \boldsymbol{B}$ from classification $\boldsymbol{A}$ to classification $\boldsymbol{B}$.

```
(7) (KIF$function infomorphism)
    (= (KIF$source infomorphism) bond)
    (= (KIF$target infomorphism) CLS.REL$infomorphism)
    (forall (?b (bond ?b))
        (and (= (CLS.REL$source (infomorphism ?b)) (source ?b))
             (= (CLS.REL$target (infomorphism ?b)) (target ?b))
             (= (CLS.REL$instance (infomorphism ?b))
                (REL$right-residuation [(classification ?b) (source ?b)]))
             (= (CLS.REL$type (infomorphism ?b))
                (REL$left-residuation [(classification ?b) (target ?b)]))))
```

- The fundamental closure property implies that the bond of this relational infomorphism is the original bond.

```
    (forall (?b (bond ?b))
        (= (CLS.REL$bond (infomorphism ?b)) ?b))
```

- The fundamental closure property of a bond $\boldsymbol{F} : \boldsymbol{A} \rightharpoonup \boldsymbol{B}$, the type-closure of $\boldsymbol{F}$ with respect to $\boldsymbol{A}$ and the instance-closure of $\boldsymbol{F}$ with respect to $\boldsymbol{B}$, can also be expressed as follows.

```
    (forall (?b (bond ?b))
        (and (CLS.CL$type-closed (classification ?b) (source ?b))
             (CLS.CL$instance-closed (classification ?b) (target ?b))))
```

- Associated with any bond is an *adjoint pair* between the complete lattices of source and target classifications.

- Since $\boldsymbol{F}$ is type-closed with respect to $\boldsymbol{A}$, there is an associated *coreflection* (adjoint pair) $\mathit{corefl}_{\boldsymbol{A},\boldsymbol{F}} = \langle \partial_0, \tilde{\partial}_0 \rangle : \mathit{lat}(\boldsymbol{F}) \rightleftarrows \mathit{lat}(\boldsymbol{A})$ between their complete lattices, where $\tilde{\partial}_0 : \mathit{lat}(\boldsymbol{A}) \to \mathit{lat}(\boldsymbol{F})$ is right adjoint right inverse (rari) to $\partial_0 : \mathit{lat}(\boldsymbol{F}) \to \mathit{lat}(\boldsymbol{A})$.
- Since $\boldsymbol{F}$ is instance-closed with respect to $\boldsymbol{B}$, there is an associated *reflection* (adjoint pair) $\mathit{refl}_{\boldsymbol{F},\boldsymbol{B}} = \langle \tilde{\partial}_1, \partial_1 \rangle : \mathit{lat}(\boldsymbol{B}) \rightleftarrows \mathit{lat}(\boldsymbol{F})$ between their complete lattices, where $\tilde{\partial}_1 : \mathit{lat}(\boldsymbol{B}) \to \mathit{lat}(\boldsymbol{F})$ is left adjoint right inverse (lari) to $\partial_1 : \mathit{lat}(\boldsymbol{F}) \to \mathit{lat}(\boldsymbol{B})$.

The adjoint pair is the composition of the reflection followed by the coreflection.

```
(8) (KIF$function adjoint-pair)
    (= (KIF$source adjoint-pair) bond)
    (= (KIF$target adjoint-pair) LAT.ADJ$adjoint-pair)
    (forall (?b (bond ?b))
        (and (= (LAT.ADJ$source (adjoint-pair ?b))
                (CLS.CL$complete-lattice (target ?b)))
             (= (LAT.ADJ$target (adjoint-pair ?b))
                (CLS.CL$complete-lattice (source ?b)))
             (= (adjoint-pair ?b)
                (LAT.ADJ$composition
                    (CLS.CL$reflection [(target ?b) (classification ?b)])
                    (CLS.CL$coreflection [(classification ?b) (source ?b)])))))
```

- Two bonds $\boldsymbol{F}_1 : \boldsymbol{A} \rightharpoonup \boldsymbol{B}$ and $\boldsymbol{F}_2 : \boldsymbol{B} \rightharpoonup \boldsymbol{C}$ are *composable* when the target classification of the first is the source classification of the second. The *composition* of two composable bonds is the bond $\boldsymbol{F}_1 \circ \boldsymbol{F}_2 \triangleq (\boldsymbol{B}/\boldsymbol{F}_2)\backslash\boldsymbol{F}_1 : \boldsymbol{A} \rightharpoonup \boldsymbol{C}$ defined using left and right residuation. Since both $\boldsymbol{F}$ and $\boldsymbol{G}$ being bonds are closed with respect to $\boldsymbol{B}$, an equivalent expression for the composition is $\boldsymbol{F}_1 \circ \boldsymbol{F}_2 \triangleq \boldsymbol{F}_2/(\boldsymbol{F}_1\backslash\boldsymbol{B}) : \boldsymbol{A} \rightharpoonup \boldsymbol{C}$. Pointwise, the composition is $\boldsymbol{F}_1 \circ \boldsymbol{F}_2 = \{(c, \alpha) \mid \boldsymbol{F}_1\alpha \supseteq (c\boldsymbol{F}_2)^{\boldsymbol{B}}\}$. To check closure, $(\boldsymbol{A}/(\boldsymbol{F}_1 \circ \boldsymbol{F}_2))\backslash\boldsymbol{A} = (\boldsymbol{A}/((\boldsymbol{B}/\boldsymbol{F}_2)\backslash\boldsymbol{F}_1))\backslash\boldsymbol{A} = (\boldsymbol{B}/\boldsymbol{F}_2)\backslash\boldsymbol{F}_1$, since $\boldsymbol{F}_1$ being a collective $\boldsymbol{A}$-intent means that $(\boldsymbol{B}/\boldsymbol{F}_2)\backslash\boldsymbol{F}_1$ is also a collective $\boldsymbol{A}$-intent.

The adjoint pair associated with a bond is the image of the morphism function of the complete adjoint functor applied to the bond:

$A$ : BOND $\to$ COMPLETE ADJOINT.

```
(8) (KIF$opspan composable-opspan)
    (= composable-opspan [target source])

(9) (KIF$relation composable)
    (= (KIF$collection1 composable) bond)
    (= (KIF$collection2 composable) bond)
    (= (KIF$extent composable) (KIF$pullback composable-opspan))

(11) (KIF$function composition)
     (= (KIF$source composition) (KIF$pullback composable-opspan))
     (= (KIF$target composition) bond)
     (forall (?b1 (bond ?b1) ?b2 (bond ?b2) (composable ?b1 ?b2))
       (and (= (source (composition [?b1 ?b2])) (source ?b1))
            (= (target (composition [?b1 ?b2])) (target ?b2))
            (= (classification (composition [?b1 ?b2]))
               (REL$left-residuation [(REL$right-residuation [?b2 (source ?b2)]) ?b1]))))
```

- With respect to bond composition, the *identity* bond at any classification $\boldsymbol{A}$ is $\boldsymbol{A}$.

```
(12) (KIF$function identity)
     (= (KIF$source identity) CLS$classification)
     (= (KIF$target identity) bond)
     (forall (?a (CLS$classification ?a))
         (and (= (source (identity ?a)) ?a)
              (= (target (identity ?a)) ?a)
              (= (classification (identity ?a)) ?a)))
```

- For any given bond $\boldsymbol{F} : \boldsymbol{A} \rightharpoonup \boldsymbol{B}$ the *dual bond* $\boldsymbol{F}^{\mathrm{op}} : \boldsymbol{B}^{\mathrm{op}} \rightharpoonup \boldsymbol{A}^{\mathrm{op}}$ is the bond with source and target classifications switched, and source, target and classification relations transposed.

```
(13) (KIF$function opposite)
     (= (KIF$source opposite) bond)
     (= (KIF$target opposite) bond)
     (forall (?f (bond ?f))
         (and (= (source (opposite ?f)) (CLS$opposite (target ?f)))
              (= (target (opposite ?f)) (CLS$opposite (source ?f)))
              (= (classification (opposite ?f)) (CLS$opposite (classification ?f)))))
```

The fundamental property of bonds for composition, identity and involution follow from basic properties of residuation.

- The basic theorem of Formal Concept Analysis can be framed in terms of two fundamental bonds between any classification and its associated concept lattice.
  - For any classification $\boldsymbol{A}$ there is an *instance embedding* bond $\iota_{\boldsymbol{A}} : \mathsf{L}(\boldsymbol{A}) \rightharpoonup \boldsymbol{A}$, whose source classification is the classification of the complete lattice of $\boldsymbol{A}$ and whose target classification is $\boldsymbol{A}$. The classification of the instance embedding bond is the instance embedding classification. The pair $\langle \mathsf{L}(\boldsymbol{A})/\iota_{\boldsymbol{A}}, \tau_{\boldsymbol{A}} \rangle : \mathsf{L}(\boldsymbol{A}) \rightrightarrows \boldsymbol{A}$ is a relational infomorphism whose bond is the instance embedding bond.
  - For any classification $\boldsymbol{A}$ the *type embedding* bond $\tau_{\boldsymbol{A}} : \boldsymbol{A} \rightharpoonup \mathsf{L}(\boldsymbol{A})$, whose source classification is $\boldsymbol{A}$ and whose target classification is the classification of the complete lattice of $\boldsymbol{A}$. The classification of the type embedding bond is the type embedding classification. The pair $\langle \iota_{\boldsymbol{A}}, \tau_{\boldsymbol{A}} \backslash \mathsf{L}(\boldsymbol{A}) \rangle : \boldsymbol{A} \rightrightarrows \mathsf{L}(\boldsymbol{A})$ is a relational infomorphism whose bond is the type embedding bond.

```
(14) (KIF$function iota)
     (= (KIF$source iota) CLS$classification)
     (= (KIF$target iota) bond)
     (forall (?a (CLS$classification ?a))
         (and (= (source (iota ?a)) (LAT$classification (CLS.CL$complete-lattice ?a)))
              (= (target (iota ?a)) ?a)
              (= (classification (iota ?a)) (CLS.CL$iota ?a))))

(16) (KIF$function tau)
     (= (KIF$source tau) CLS$classification)
     (= (KIF$target tau) bond)
     (forall (?a (CLS$classification ?a))
         (and (= (source (iota ?a)) ?a)
              (= (target (iota ?a)) (LAT$classification (CLS.CL$complete-lattice ?a)))
              (= (classification (tau ?a)) (CLS.CL$tau ?a))))
```

- The instance and type embedding bonds are inverse to each other: $\iota_{\boldsymbol{A}} \mathbin{\square} \tau_{\boldsymbol{A}} = Id_{\mathsf{L}(\boldsymbol{A})}$ and $\tau_{\boldsymbol{A}} \mathbin{\square} \iota_{\boldsymbol{A}} = Id_{\boldsymbol{A}}$.

```
     (forall (?a (CLS$classification ?a))
         (and (= (composition [(iota ?a) (tau ?a)])
                 (identity (LAT$classification (CLS.CL$complete-lattice ?a))))
              (= (composition [(tau ?a) (iota ?a)])
                 (identity ?a))))
```

- This demonstrates the bond isomorphism (commuting Diagram 5) between any classification $\boldsymbol{A}$ and the classification of the complete lattice of $\boldsymbol{A}$. It can be proven that this is a natural isomorphism; that is, that the quasi-category CLASSIFICATION of classifications and bonds is categorically equivalent to the quasi-category of complete lattices and adjoint pairs. Let $\boldsymbol{F} : \boldsymbol{A} \rightharpoonup \boldsymbol{B}$ be a bond with associated complete adjoint $\mathit{adj}(\boldsymbol{F}) : \mathit{lat}(\boldsymbol{B}) \rightleftarrows \mathit{lat}(\boldsymbol{A})$. The classification of the bond $\mathit{bnd}(\mathit{adj}(\boldsymbol{F})) : \mathit{cls}(\mathit{lat}(\boldsymbol{A})) \rightharpoonup \mathit{cls}(\mathit{lat}(\boldsymbol{B}))$ contains a conceptual pair $(\boldsymbol{a}, \boldsymbol{b})$ of the form $\boldsymbol{a} = (A, \Gamma)$ and $\boldsymbol{b} = (B, \Delta)$ iff $B \circ \Gamma \subseteq \boldsymbol{F}$, where $B \circ \Gamma = B \times \Gamma$ a Cartesian product or rectangle, iff $B \subseteq \Gamma^{\boldsymbol{F}}$ iff $\Gamma \subseteq B^{\boldsymbol{F}}$. So, $(\iota_{\boldsymbol{A}} \mathbin{\square} \boldsymbol{F}) \mathbin{\square} \tau_{\boldsymbol{B}} = (\boldsymbol{F}/(\iota_{\boldsymbol{A}} \backslash \boldsymbol{A})) \mathbin{\square} \tau_{\boldsymbol{B}} = (\boldsymbol{F}/\tau_{\boldsymbol{A}}) \mathbin{\square} \tau_{\boldsymbol{B}} = (\boldsymbol{B}/\tau_{\boldsymbol{B}}) \backslash (\boldsymbol{F}/\tau_{\boldsymbol{A}}) = \iota_{\boldsymbol{B}} \backslash (\boldsymbol{F}/\tau_{\boldsymbol{A}}) = \mathit{bnd}(\mathit{adj}(\boldsymbol{F}))$, by bond composition and properties of the instance and type relations. Hence, $\mathit{bnd}(\mathit{adj}(\boldsymbol{F})) \mathbin{\square} \iota_{\boldsymbol{B}} = \iota_{\boldsymbol{A}} \mathbin{\square} \boldsymbol{F}$. This

$$\begin{tikzcd}
\mathsf{B}(\mathsf{A}(\boldsymbol{A})) \arrow[r, harpoon, "\iota_{\boldsymbol{A}}"] \arrow[d, harpoon, "\mathsf{B}(\mathsf{A}(\boldsymbol{F}))"] & \boldsymbol{A} \arrow[r, harpoon, "\tau_{\boldsymbol{A}}"] \arrow[d, harpoon, "\boldsymbol{F}"] & \mathsf{B}(\mathsf{A}(\boldsymbol{A})) \arrow[d, harpoon, "\mathsf{B}(\mathsf{A}(\boldsymbol{F}))"] \\
\mathsf{B}(\mathsf{A}(\boldsymbol{B})) \arrow[r, harpoon, "\iota_{\boldsymbol{B}}"'] & \boldsymbol{B} \arrow[r, harpoon, "\tau_{\boldsymbol{B}}"'] & \mathsf{B}(\mathsf{A}(\boldsymbol{B}))
\end{tikzcd}$$

**Diagram 5: Natural Isomorphism**

For any classification $\boldsymbol{A}$ the iota bond $\iota_{\boldsymbol{A}}$ is the $\boldsymbol{A}^{\text{th}}$ component of a natural isomorphism $\mathsf{B}(\mathsf{A}(\boldsymbol{A})) \cong \boldsymbol{A}$, demonstrating that the quasi-category of bonds is categorically equivalent to the quasi-category of complete adjoints:

COMPLETE ADJOINT ≡ BOND.

sition and properties of the instance and type relations. Hence, $bnd(adj(\boldsymbol{F})) \circ \iota_{\boldsymbol{B}} = \iota_{\boldsymbol{A}} \circ \boldsymbol{F}$. This proves the required naturality condition.

```
      (forall (?b (bond ?b))
          (= (composition [(CL.ADJ$bond (adjoint-pair ?b)) (iota (target ?f))])
             (composition [(iota (source ?f)) ?f])))
```

## Bonding Pairs

**CLS.BNDPR**

The bond equivalent to a complete homomorphism would seem to be given by two bonds $\boldsymbol{F} : \boldsymbol{A} \rightharpoonup \boldsymbol{B}$ and $\boldsymbol{G} : \boldsymbol{B} \rightharpoonup \boldsymbol{A}$ where the right adjoint $\psi_{\boldsymbol{F}} : \boldsymbol{L}(\boldsymbol{A}) \rightarrow \boldsymbol{L}(\boldsymbol{B})$ of the complete adjoint $\boldsymbol{A}(\boldsymbol{F}) = \langle \varphi_{\boldsymbol{F}}, \psi_{\boldsymbol{F}} \rangle : \boldsymbol{L}(\boldsymbol{A}) \rightleftarrows \boldsymbol{L}(\boldsymbol{B})$ of one bond (say $\boldsymbol{F}$, without loss of generality) is equal to the left adjoint $\varphi_{\boldsymbol{G}} : \boldsymbol{L}(\boldsymbol{A}) \rightarrow \boldsymbol{L}(\boldsymbol{B})$ of the complete adjoint $\boldsymbol{A}(\boldsymbol{G}) = \langle \varphi_{\boldsymbol{G}}, \psi_{\boldsymbol{G}} \rangle : \boldsymbol{L}(\boldsymbol{B}) \rightleftarrows \boldsymbol{L}(\boldsymbol{A})$ of the other bond $\boldsymbol{G}$ with the resultant adjunctions, $\varphi_{\boldsymbol{F}} \dashv \psi_{\boldsymbol{F}} = \varphi_{\boldsymbol{G}} \dashv \psi_{\boldsymbol{G}}$, where the middle adjoint is the complete homomorphism. This is indeed the case, but the question is what constraint to place on $\boldsymbol{F}$ and $\boldsymbol{G}$ in order for this to hold. The simple answer is to identify the actions of the two monotonic functions $\psi_{\boldsymbol{G}}$ and $\varphi_{\boldsymbol{F}}$. Let $(A, \Gamma) \in \boldsymbol{L}(\boldsymbol{A})$ be any formal concept in $\boldsymbol{L}(\boldsymbol{A})$. The action of the left adjoint $\varphi_{\boldsymbol{G}}$ on this concept is $(A, \Gamma) \mapsto (A^{\boldsymbol{GB}}, A^{\boldsymbol{G}})$, whereas the action of the right adjoint $\psi_{\boldsymbol{F}}$ on this concept is $(A, \Gamma) \mapsto (\Gamma^{\boldsymbol{F}}, \Gamma^{\boldsymbol{FB}})$. So the appropriate pointwise constraints are: $A^{\boldsymbol{GB}} = \Gamma^{\boldsymbol{F}}$ and $\Gamma^{\boldsymbol{FB}} = A^{\boldsymbol{G}}$, for every concept $(A, \Gamma) \in \boldsymbol{L}(\boldsymbol{A})$. The relational representation of these *pointwise constraints* is used in the definition of a bonding pair.

$$\begin{tikzcd}
\boldsymbol{L}(\boldsymbol{A}) \arrow[r, rightharpoonup, "\iota_{\boldsymbol{A}}"] & \boldsymbol{A} \arrow[d, rightharpoondown, "\boldsymbol{F}"] \\
\boldsymbol{A} \arrow[u, rightharpoonup, "\tau_{\boldsymbol{B}}"] & \boldsymbol{B} \arrow[l, leftharpoondown, "\boldsymbol{G}"']
\end{tikzcd}$$

**Diagram 6: Bonding Pair**

○ A *bonding pair* $\langle \boldsymbol{F}, \boldsymbol{G} \rangle : \boldsymbol{A} \rightleftharpoons \boldsymbol{B}$ between two classifications $\boldsymbol{A}$ and $\boldsymbol{B}$ (Figure 15) is a contravariant pair of bonds, a bond $\boldsymbol{F} : \boldsymbol{A} \rightharpoonup \boldsymbol{B}$ in the *forward* direction and a bond $\boldsymbol{G} : \boldsymbol{A} \leftharpoondown \boldsymbol{B}$ in the *reverse* direction, satisfying the following *pairing constraints*:

$$\boldsymbol{F}/\tau_{\boldsymbol{A}} = \boldsymbol{B}/(\iota_{\boldsymbol{A}}\backslash\boldsymbol{G}) \text{ and } \iota_{\boldsymbol{A}}\backslash\boldsymbol{G} = (\boldsymbol{F}/\tau_{\boldsymbol{A}})\backslash\boldsymbol{B},$$

$$\begin{tikzcd}
 & typ(\boldsymbol{A}) & typ(\boldsymbol{B}) \\
\boldsymbol{L}(\boldsymbol{A}) \arrow[ur, "\tau_{\boldsymbol{A}}"] \arrow[urr, "\iota_{\boldsymbol{A}}\backslash\boldsymbol{G}"'] & & \\
 & inst(\boldsymbol{A}) \arrow[uu] \arrow[ul, "\iota_{\boldsymbol{A}}"] \arrow[uur, "\boldsymbol{G}"'] & inst(\boldsymbol{B}) \arrow[uu] \arrow[uul, "\boldsymbol{F}"'] \arrow[ull, "\boldsymbol{F}/\tau_{\boldsymbol{A}}"]
\end{tikzcd}$$

**Figure 15: Bonding Pair**

which state that $(\boldsymbol{F}/\tau_{\boldsymbol{A}}, \iota_{\boldsymbol{A}}\backslash\boldsymbol{G}) = (\iota_{\boldsymbol{A}} \circ \boldsymbol{F}, \boldsymbol{G} \circ \tau_{\boldsymbol{B}})$ is an $\boldsymbol{L}(\boldsymbol{A})$-indexed collective $\boldsymbol{B}$-concept (Diagram 6). The definitions of the relations $\boldsymbol{F}/\tau_{\boldsymbol{A}}$ and $\iota_{\boldsymbol{A}}\backslash\boldsymbol{G}$ are given as follows: $\boldsymbol{F}/\tau_{\boldsymbol{A}} = \{(b, \boldsymbol{a}) \mid int(\boldsymbol{a}) \subseteq b\boldsymbol{F}\} = \{(b, \boldsymbol{a}) \mid ((b\boldsymbol{F})^{\boldsymbol{A}}, b\boldsymbol{F}) \leq_{\boldsymbol{B}} \boldsymbol{a}\}$ and $\iota_{\boldsymbol{A}}\backslash\boldsymbol{G} = \{(\boldsymbol{a}, \beta) \mid ext(\boldsymbol{a}) \subseteq \boldsymbol{G}\beta\} = \{(\boldsymbol{a}, \beta) \mid \boldsymbol{a} \leq_{\boldsymbol{B}} (\boldsymbol{G}\beta, (\boldsymbol{G}\beta)^{\boldsymbol{A}})\}$. Any concept $\boldsymbol{a} = (A, \Gamma) \in \boldsymbol{L}(\boldsymbol{A})$ is mapped by the relations as: $(\boldsymbol{F}/\tau_{\boldsymbol{A}})((A, \Gamma)) = \{b \mid \Gamma \subseteq b\boldsymbol{F}\} = \Gamma^{\boldsymbol{F}}$ and $(\iota_{\boldsymbol{A}}\backslash\boldsymbol{G})((A, \Gamma)) = \{\beta \mid A \subseteq \boldsymbol{G}\beta\} = A^{\boldsymbol{G}}$. Hence, pointwise the constraints are $\Gamma^{\boldsymbol{F}} = B^{\boldsymbol{GB}}$ and $A^{\boldsymbol{G}} = \Gamma^{\boldsymbol{FB}}$. These are the pointwise constraints discussed above. Let BONDING PAIR denote the quasi-category of classifications and bonding pairs.

```
(1) (KIF$collection bonding-pair)

(2) (KIF$function source)
    (= (KIF$source source) bonding-pair)
    (= (KIF$target source) CLS$classification)

(3) (KIF$function target)
    (= (KIF$source target) bonding-pair)
    (= (KIF$target target) CLS$classification)

(4) (KIF$function forward)
    (= (KIF$source forward) bonding-pair)
    (= (KIF$target forward) bond)
    (forall (?bp (bonding-pair ?bp))
        (and (= (CLS.BND$source (forward ?bp)) (source ?bp))
             (= (CLS.BND$target (forward ?bp)) (target ?bp))))

(5) (KIF$function reverse)
    (= (KIF$source reverse) bonding-pair)
    (= (KIF$target reverse) bond)
    (forall (?bp (bonding-pair ?bp))
        (and (= (CLS.BND$source (reverse ?bp)) (target ?bp))
             (= (CLS.BND$target (reverse ?bp)) (source ?bp))))

(6) (forall (?bp (bonding-pair ?bp))
        (and (= (REL$right-residuation [(CLS.CL$tau (source ?bp)) (forward ?bp)])
                (REL$right-residuation
                    [(REL$left-residuation [(CLS.CL$iota (source ?bp)) (reverse ?bp)])
                     (target ?bp)]))
             (= (REL$left-residuation [(CLS.CL$iota (source ?bp)) (reverse ?bp)])
                (REL$left-residuation
                    [(REL$right-residuation [(CLS.CL$tau (source ?bp)) (forward ?bp)])
                     (target ?bp)]))))
```

o The pointwise constraints can be lifted to a collective setting – any bonding pair $\langle \boldsymbol{F}, \boldsymbol{G}\rangle : \boldsymbol{A} \rightleftharpoons \boldsymbol{B}$ preserves collective concepts: for any $A$-indexed collective $\boldsymbol{A}$-concept $(X, Y)$, $X\backslash\boldsymbol{A} = Y$ and $\boldsymbol{A}/Y = X$, the *conceptual image* $(\boldsymbol{F}/Y, X\backslash\boldsymbol{G})$ is an $A$-indexed collective $\boldsymbol{B}$-concept (Figure 16), since $\boldsymbol{B}/(X\backslash\boldsymbol{G}) = \boldsymbol{F}/Y$ and $(\boldsymbol{F}/Y)\backslash\boldsymbol{B} = X\backslash\boldsymbol{G}$. An important special case is the $L(\boldsymbol{A})$-indexed collective $\boldsymbol{A}$-concept $(\iota_{\boldsymbol{A}}, \tau_{\boldsymbol{A}})$. To state that the $\langle \boldsymbol{F}, \boldsymbol{G}\rangle$-image $(\boldsymbol{F}/\tau_{\boldsymbol{A}}, \iota_{\boldsymbol{A}}\backslash\boldsymbol{F})$ is an $L(\boldsymbol{A})$-indexed collective $\boldsymbol{B}$-concept, is to assert the pairing constraints $\boldsymbol{F}/\tau_{\boldsymbol{A}} = \boldsymbol{B}/(\iota_{\boldsymbol{A}}\backslash\boldsymbol{G})$ and $\iota_{\boldsymbol{A}}\backslash\boldsymbol{G} = (\boldsymbol{F}/\tau_{\boldsymbol{B}})\backslash\boldsymbol{B}$ of the bonding pair. So, the concise definition in terms of pairing constraints, the original pointwise definition above, and the assertion that $\langle \boldsymbol{F}, \boldsymbol{G}\rangle$ preserves all collective concepts, are equivalent versions of the notion of a bonding pair.

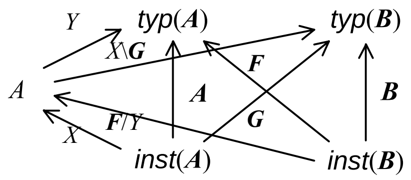


**Figure 16: Conceptual Image**

```
(7) (KIF$function conceptual-image)
    (= (KIF$source conceptual-image) bonding-pair)
    (= (KIF$target conceptual-image) KIF$function)
    (forall (?bp (bonding-pair ?bp))
        (and (= (KIF$source (conceptual-image ?bp)) (CLS.FIB$concept (source ?bp)))
             (= (KIF$target (conceptual-image ?bp)) (CLS.FIB$concept (target ?bp)))
             (forall (?a ((CLS.FIB$concept (source ?bp)) ?a))
                 (and (= ((CLS.FIB$index (target ?bp)) ((conceptual-image ?bp) ?a))
                         ((CLS.FIB$index (source ?bp)) ?a))
                      (= ((CLS.FIB$extent (target ?bp)) ((conceptual-image ?bp) ?a))
                         (CLS.FIB$right-residuation
                             [(((CLS.FIB$intent (source ?bp)) ?a)
                              (CLS.BND$classification (forward ?bp))]))
                      (= ((CLS.FIB$intent (target ?bp)) ((conceptual-image ?bp) ?a))
                         (CLS.FIB$left-residuation
                             [(((CLS.FIB$extent (source ?bp)) ?a)
                              (CLS.BND$classification (reverse ?bp))]))))))
```

○ Let $\langle \boldsymbol{F}, \boldsymbol{G}\rangle : \boldsymbol{A} \rightleftharpoons \boldsymbol{B}$ be any bonding pair. Then $\boldsymbol{F} : \boldsymbol{A} \rightharpoonup \boldsymbol{B}$ is a bond in the forward direction from classification $\boldsymbol{A}$ to classification $\boldsymbol{B}$, and $\boldsymbol{G} : \boldsymbol{A} \leftharpoondown \boldsymbol{B}$ is a bond in the reverse direction to classification $\boldsymbol{A}$ from classification $\boldsymbol{B}$. Applying the complete adjoint functor $A$ : BOND → COMPLETE ADJOINT, we get two adjoint pairs in opposite directions: an adjoint pair $\langle \varphi_F, \psi_F\rangle : L(\boldsymbol{B}) \rightleftarrows L(\boldsymbol{A})$ in the forward direction and an adjoint pair $\langle \varphi_G, \psi_G\rangle : L(\boldsymbol{A}) \rightleftarrows L(\boldsymbol{B})$ in the reverse direction. The meet-preserving monotonic function $\psi_F : L(\boldsymbol{A}) \rightarrow L(\boldsymbol{B})$ is equal to the join-preserving monotonic function $\varphi_G : L(\boldsymbol{A}) \rightarrow L(\boldsymbol{B})$, giving a complete lattice *homomorphism*. This function is the unique mediating function for the $L(\boldsymbol{A})$-indexed collective $\boldsymbol{B}$-concept $(\boldsymbol{F}/\tau_{\boldsymbol{A}}, \iota_{\boldsymbol{A}}\backslash\boldsymbol{G})$, the $\langle \boldsymbol{F}, \boldsymbol{G}\rangle$-image of the $L(\boldsymbol{A})$-indexed collective $\boldsymbol{A}$-concept $(\iota_{\boldsymbol{A}}, \tau_{\boldsymbol{A}})$, whose closure expressions define the pairing constraints.

The complete lattice homomorphism associated with a bonding pair is the image of the morphism function of the complete lattice functor applied to the bonding pair:

$A^2$ : BONDING PAIR → COMPLETE LATTICE.

```
(8) (KIF$function homomorphism)
    (= (KIF$source homomorphism) bonding-pair)
    (= (KIF$target homomorphism) CL.MOR$homomorphism)
```

```
        (forall (?bp (bonding-pair ?bp))
            (and (= (CL.MOR$source (homomorphism ?bp))
                    (CLS.CL$complete-lattice (source ?bp)))
                 (= (CL.MOR$target (homomorphism ?bp))
                    (CLS.CL$complete-lattice (target ?bp)))
                 (= (CL.MOR$forward (homomorphism ?bp))
                    (CLS.BND$adjoint-pair (forward ?h)))
                 (= (CL.MOR$reverse (homomorphism ?bp))
                    (CLS.BND$adjoint-pair (reverse ?h)))))
```

- Two bonding pairs $\langle \boldsymbol{F}, \boldsymbol{G} \rangle : \boldsymbol{A} \rightleftharpoons \boldsymbol{B}$ and $\langle \boldsymbol{M}, \boldsymbol{N} \rangle : \boldsymbol{B} \rightleftharpoons \boldsymbol{C}$ are *composable* when the target of the first is the source of the second. The *composition* $\langle \boldsymbol{F}, \boldsymbol{G} \rangle \circ \langle \boldsymbol{M}, \boldsymbol{N} \rangle \triangleq \langle \boldsymbol{F} \circ \boldsymbol{M}, \boldsymbol{N} \circ \boldsymbol{G} \rangle : \boldsymbol{A} \rightleftharpoons \boldsymbol{C}$ of two composable bonding pairs is defined in terms of bond composition.

```
(9) (KIF$opspan composable-opspan)
    (= composable-opspan [target source])

(10) (KIF$relation composable)
     (= (KIF$collection1 composable) bonding-pair)
     (= (KIF$collection2 composable) bonding-pair)
     (= (KIF$extent composable) (KIF$pullback composable-opspan))

(11) (KIF$function composition)
     (= (KIF$source composition) (KIF$pullback composable-opspan))
     (= (KIF$target composition) bonding-pair)
     (forall (?bp1 (bonding-pair ?bp1)
              ?bp2 (bonding-pair ?bp2) (composable ?bp1 ?bp2))
         (and (= (source (composition [?bp1 ?bp2])) (source ?bp1))
              (= (target (composition [?bp1 ?bp2])) (target ?bp2))
              (= (forward (composition [?bp1 ?bp2]))
                 (CLS.BND$composition (forward ?bp1) (forward ?bp2)))
              (= (reverse (composition [?bp1 ?bp2]))
                 (CLS.BND$composition [(reverse ?bp2) (reverse ?bp1)]))))
```

- For any classification $\boldsymbol{A}$, define the bonding pair *identity* $\langle Id_A, Id_A \rangle : \boldsymbol{A} \rightleftharpoons \boldsymbol{A}$ in terms of bond identity.

```
(12) (KIF$function identity)
     (= (KIF$source identity) CLS$classification)
     (= (KIF$target identity) bonding-pair)
     (forall (?a (CLS$classification ?a))
         (and (= (source (identity ?a)) ?a)
              (= (target (identity ?a)) ?a)
              (= (forward (identity ?a)) (CLS.BND$identity ?a))
              (= (reverse (identity ?a)) (CLS.BND$identity ?a))))
```

- For any classification $\boldsymbol{A}$ the type and instance embedding relations form bonding pairs in two different ways, $\langle \tau_A, \iota_A \rangle : \boldsymbol{A} \rightleftharpoons \mathsf{L}(\boldsymbol{A})$ and $\langle \iota_A, \tau_A \rangle : \mathsf{L}(\boldsymbol{A}) \rightleftharpoons \boldsymbol{A}$.

```
(13) (KIF$function tau-iota)
     (= (KIF$source tau-iota) CLS$classification)
     (= (KIF$target tau-iota) bonding-pair)
     (forall (?a (CLS$classification ?a))
         (and (= (source (tau-iota ?a)) ?a)
              (= (target (tau-iota ?a))
                 (LAT$classification (CLS.CL$complete-lattice ?a)))
              (= (forward (tau-iota ?a)) (CLS.BND$tau ?a))
              (= (reverse (tau-iota ?a)) (CLS.BND$iota ?a))))

(14) (KIF$function iota-tau)
     (= (KIF$source iota-tau) CLS$classification)
     (= (KIF$target iota-tau) bonding-pair)
     (forall (?a (CLS$classification ?a))
         (and (= (source (iota-tau ?a))
                 (LAT$classification (CLS.CL$complete-lattice ?a)))
              (= (target (iota-tau ?a)) ?a)
              (= (forward (iota-tau ?a)) (CLS.BND$iota ?a))
              (= (reverse (iota-tau ?a)) (CLS.BND$tau ?a))))
```

- For any classification $\boldsymbol{A}$ the two bonding pairs, $\langle \tau_A, \iota_A \rangle : \boldsymbol{A} \rightleftharpoons \mathsf{L}(\boldsymbol{A})$ and $\langle \iota_A, \tau_A \rangle : \mathsf{L}(\boldsymbol{A}) \rightleftharpoons \boldsymbol{A}$, are inverse to each other: $\langle \tau_A, \iota_A \rangle \circ \langle \iota_A, \tau_A \rangle = Id_A$ and $\langle \iota_A, \tau_A \rangle \circ \langle \tau_A, \iota_A \rangle = Id_{\mathsf{L}(A)}$. Therefore, each classification is isomorphic in the quasi-category BONDING PAIR to the classification of its complete lattice: $\boldsymbol{A} \cong \mathit{cls}(\mathit{clat}(\boldsymbol{A}))$.

For any classification $\boldsymbol{A}$ the iota-tau bonding pair $\langle \iota_A, \tau_A \rangle$ is the $\boldsymbol{A}$th component of a natural isomorphism $B^2(A^2(\boldsymbol{A})) \cong \boldsymbol{A}$, demonstrating that the quasi-category of bonding pairs is categorically equivalent to the quasi-category of complete lattices:

BONDING PAIR ≡ COMPLETE LATTICE.

```
    (forall (?a (CLS$classification ?a))
        (and (= (composition [(tau-iota ?a) (iota-tau ?a)])
                (identity ?a))
             (= (composition [(iota-tau ?a) (tau-iota ?a)])
                (identity (LAT$classification (CLS.CL$complete-lattice ?a))))))
```

- This is a natural isomorphism. Any bonding pair $\langle F, G \rangle : \boldsymbol{A} \rightleftharpoons \boldsymbol{B}$ satisfies the following naturality condition: $\langle \iota_A, \tau_A \rangle \circ \langle F, G \rangle = \mathit{bndpr}(\mathit{homo}(\langle F, G \rangle)) \circ \langle \iota_B, \tau_B \rangle$.

```
    (forall (?bp (bonding-pair ?bp))
        (= (composition [(iota-tau (source ?bp)) ?bp])
           (composition
               [(LAT.MOR$bonding-pair (homomorphism ?bp))
                (iota-tau (target ?bp))])))
```

- For any given bonding pair $\langle \boldsymbol{F}, \boldsymbol{G} \rangle : \boldsymbol{A} \rightleftharpoons \boldsymbol{B}$ the *dual* or *opposite* bonding pair $\langle \boldsymbol{G}^{\mathrm{op}}, \boldsymbol{F}^{\mathrm{op}} \rangle : \boldsymbol{A}^{\mathrm{op}} \rightleftharpoons \boldsymbol{B}^{\mathrm{op}}$ is the bonding pair with source/target classifications dualized, and forward/reverse bonds switched and dualized.

```
(15) (KIF$function opposite)
     (= (KIF$source opposite) bonding-pair)
     (= (KIF$target opposite) bonding-pair)
     (forall (?bp (bonding-pair ?bp))
         (and (= (source (opposite ?bp)) (CLS$opposite (source ?bp)))
              (= (target (opposite ?bp)) (CLS$opposite (target ?bp)))
              (= (forward (opposite ?bp)) (CLS.BND$opposite (reverse ?bp)))
              (= (reverse (opposite ?bp)) (CLS.BND$opposite (forward ?bp)))))
```

## *Finite Colimits*

**`CLS.COL`**

Classifications can be fused together and internalized using colimit operations. Here we present axioms that make CLASSIFICATION, the quasi-category of classifications and infomorphisms, finitely cocomplete. We assert the existence of initial classifications, binary coproducts of classifications, coequalizers of parallel pairs of infomorphisms and pushouts of spans of infomorphisms. Because of commonality, the terminology for binary coproducts, coequalizers and pushouts are put into sub-namespaces. The *diagrams* and *colimits* are denoted by both generic and specific terminology.

$INST^{-1}(\mathsf{SET}) = \mathsf{CLASSIFICATION} = TYP^{-1}(\mathsf{SET})$

$0 \dashv INST \dashv \wp$ $\wp \dashv TYP \dashv 1$

$$\begin{tikzcd} & \mathsf{CLASSIFICATION} \arrow[dl, "INST"] \arrow[dr, "TYP"] & \\ \mathsf{SET}^{op} \arrow[ur, bend left, "0"] \arrow[ur, bend right, "\wp"'] & & \mathsf{SET} \arrow[ul, bend right, "1"'] \arrow[ul, bend left, "\wp"] \end{tikzcd}$$

**Figure 17: Fibered Span**

The following discussion refers to Figure 17 (where arrows denote functors).

The existence of colimits is mediated through the quasi-adjunction $INST \dashv \wp$ between

$INST$ : CLASSIFICATION $\rightarrow$ SET$^{op}$ the underlying instance quasi-functor, and

$\wp$ : SET$^{op}$ $\rightarrow$ CLASSIFICATION the instance power quasi-functor,

and the (trivial) quasi-adjunction $TYP \dashv 1$ between

$TYP$ : CLASSIFICATION $\rightarrow$ SET the underlying type quasi-functor, and

1 : SET $\rightarrow$ CLASSIFICATION the (trivial) terminal type quasi-functor.

Since the quasi-functors $INST$ and $TYP$ are left adjoint, they preserve all colimits. Using preservation as a guide, all diagrams, cocones and colimits in CLASSIFICATION have (and use) underlying instance/type diagrams, instance cones, type cocones, instance limits and type colimits in SET.

### The Initial Classification

○ There is a special classification $\boldsymbol{0} = \langle 1 = \{0\}, \emptyset, \emptyset\rangle$ (see Figure 18, where arrows denote functions) called the *initial classification*, which has only one instance 0, no types, and empty incidence. The initial classification has the property that for any classification $A$ = $\langle inst(A), typ(A), \models_A\rangle$ there is a *counique infomorphism* $!_A : \boldsymbol{0} \rightleftarrows A$, the unique infomorphism from $\boldsymbol{0}$ to $A$. The instance function of this infomorphism is the unique function from the instance class $inst(A)$ to the terminal (unit) class $1$. The type function of this infomorphism is the counique function (the empty function) from the initial (empty or null) class $\emptyset$ to the type class $typ(A)$. The fundamental constraint for the counique infomorphism is vacuous.

$$\begin{tikzcd} \boldsymbol{0} \arrow[r, shift left, "!_A"] & A \arrow[l, shift left] \end{tikzcd}$$

**Figure 18: Initial Classification & Universality**

```
(1) (CLS$classification initial)
    (= (CLS$instance initial) SET.LIM$terminal)
    (= (CLS$type initial) SET.COL$initial)
    (= initial (REL$empty [SET.LIM$terminal SET.COL$initial])

(2) (KIF$function counique)
    (= (KIF$source counique) CLS$classification)
    (= (KIF$target counique) CLS.INFO$infomorphism)
    (forall (?a (CLS$classification ?a))
        (and (= (CLS.INFO$source (counique ?a)) initial)
             (= (CLS.INFO$target (counique ?a)) ?a)
             (= (CLS.INFO$instance (counique ?a)) (SET.LIM$unique (CLS$instance ?s)))
             (= (CLS.INFO$type (counique ?a)) (SET.COL$counique (CLS$type ?a)))))
```

## Binary Coproducts

**`CLS.COL.COPRD`**

A *binary coproduct* (Figure 19) is a finite colimit for a diagram of shape *two* = • •. Such a diagram (of classifications and infomorphisms) is called a *pair* of classifications. Given a pair of classifications $\boldsymbol{A}_1 = \langle \mathsf{inst}(\boldsymbol{A}_1), \mathsf{typ}(\boldsymbol{A}_1), \models_1\rangle$ and $\boldsymbol{A}_2 = \langle \mathsf{inst}(\boldsymbol{A}_2), \mathsf{typ}(\boldsymbol{A}_2), \models_2\rangle$, the *coproduct* or *sum* $\boldsymbol{A}_1 + \boldsymbol{A}_2$ (see top of Figure 3, where arrows denote functions) is the classification defined as follows:

- The class of instances is the Cartesian product $\mathsf{inst}(\boldsymbol{A}_1+\boldsymbol{A}_2) = \mathsf{inst}(\boldsymbol{A}_1)\times\mathsf{inst}(\boldsymbol{A}_2)$. So, instances of $\boldsymbol{A}_1 + \boldsymbol{A}_2$ are pairs $(i_1, i_2)$ of instances $i_1 \in \mathsf{inst}(\boldsymbol{A}_1)$ and $i_2 \in \mathsf{inst}(\boldsymbol{A}_2)$.
- The class of types is the disjoint union $\mathsf{typ}(\boldsymbol{A}_1+\boldsymbol{A}_2) = \mathsf{typ}(\boldsymbol{A}_1)+\mathsf{typ}(\boldsymbol{A}_2)$. Concretely, types of $\boldsymbol{A}_1 + \boldsymbol{A}_2$ are either pairs $(1, t_1)$ where $t_1 \in \mathsf{typ}(\boldsymbol{A}_1)$ or pairs $(2, t_2)$ where $t_2 \in \mathsf{inst}(\boldsymbol{A}_2)$.
- The incidence $\models_{1+2}$ of $\boldsymbol{A}_1 + \boldsymbol{A}_2$ is defined by

  $(i_1, i_2) \models_{1+2} (1, t_1)$ when $i_1 \models_1 t_1$

  $(i_1, i_2) \models_{1+2} (2, t_2)$ when $i_2 \models_2 t_2$.

$$\begin{tikzcd}
\boldsymbol{A}_1 \arrow[r, shift left, "\iota_1"] \arrow[dr, shift left] & \boldsymbol{A}_1+\boldsymbol{A}_2 \arrow[l, shift left] \arrow[r, shift left] \arrow[d, shift left] & \boldsymbol{A}_2 \arrow[l, shift left, "\iota_2"'] \arrow[dl, shift left] \\
 & \boldsymbol{B} \arrow[ul, shift left, "f_1"] \arrow[u, shift left, "{[f_1,f_2]}"] \arrow[ur, shift left, "f_2"'] &
\end{tikzcd}$$

**Figure 19: Binary Coproduct & Universality**

- A *pair* (of classifications) is the appropriate base diagram for a binary coproduct. Each pair consists of a pair of classifications called *classification1* and *classification2*. We use either the generic term '`diagram`' or the specific term '`pair`' to denotes the *pair* collection. Pairs are determined by their two component classifications.

```
(1) (KIF$collection diagram)
    (KIF$collection pair)
    (= pair diagram)

(2) (KIF$function classification1)
    (= (KIF$source classification1) diagram)
    (= (KIF$target classification1) CLS$classification)

(3) (KIF$function classification2)
    (= (KIF$source classification2) diagram)
    (= (KIF$target classification2) CLS$classification)

    (forall (?p (diagram ?p) ?q (diagram ?q))
        (=> (and (= (classification1 ?p) (classification1 ?q))
                 (= (classification2 ?p) (classification2 ?q)))
            (= ?p ?q)))
```

- There is an *instance pair* or *instance diagram* function, which maps a pair of classifications to the underlying pair of instance classes. Similarly, there is a *type pair* function, which maps a pair of classifications to the underlying pair of type classes.

```
(4) (KIF$function instance-diagram)
    (KIF$function instance-pair)
    (= instance-pair instance-diagram)
    (= (KIF$source instance-diagram) diagram)
    (= (KIF$target instance-diagram) SET.LIM.PRD$diagram)
    (forall (?p (diagram ?p))
        (and (= (SET.LIM.PRD$class1 (instance-diagram ?p))
                (CLS$instance (classification1 ?p)))
             (= (SET.LIM.PRD$class2 (instance-diagram ?p))
                (CLS$instance (classification2 ?p)))))

(5) (KIF$function type-diagram)
    (KIF$function type-pair)
    (= type-pair type-diagram)
    (= (KIF$source type-diagram) diagram)
    (= (KIF$target type-diagram) SET.COLIM.COPRD$diagram)
    (forall (?p (diagram ?p))
        (and (= (SET.COLIM.COPRD$class1 (type-diagram ?p))
                (CLS$type (classification1 ?p)))
             (= (SET.COLIM.COPRD$class2 (type-diagram ?p))
                (CLS$type (classification2 ?p)))))
```

- Every pair has an *opposite*.

```
(6) (KIF$function opposite)
    (= (KIF$source opposite) pair)
    (= (KIF$target opposite) pair)
    (forall (?p (pair ?p))
        (and (= (classification1 (opposite ?p)) (classification2 ?p))
             (= (classification2 (opposite ?p)) (classification1 ?p))))
```

- The opposite of the opposite is the original pair – the following theorem can be proven.

```
    (forall (?p (pair ?p))
        (= (opposite (opposite ?p)) ?p))
```

**Figure 20: Binary Coproduct Cocone**

$$\begin{tikzcd}
A_1 \arrow[dr, shift left] & & A_2 \arrow[dl, shift left] \\
& A \arrow[ul, shift left, "op1st"] \arrow[ur, shift left, "op2nd"'] &
\end{tikzcd}$$

- A *binary coproduct cocone* is the appropriate cocone for a binary coproduct. A coproduct cocone (see Figure 20, where arrows denote functions) consists of a pair of infomorphisms called *opfirst* and *opsecond*. These are required to have a common source classification called the *opvertex* of the cocone. Each binary coproduct cocone is under a pair of classifications.

```
(7) (KIF$collection cocone)

(8) (KIF$function cocone-diagram)
    (= (KIF$source cocone-diagram) cocone)
    (= (KIF$target cocone-diagram) diagram)

(9) (KIF$function opvertex)
    (= (KIF$source opvertex) cocone)
    (= (KIF$target opvertex) CLS$classification)

(10) (KIF$function opfirst)
     (= (KIF$source opfirst) cocone)
     (= (KIF$target opfirst) CLS.INFO$infomorphism)
     (forall (?s (cocone ?s))
         (and (= (CLS.INFO$source (opfirst ?s))
                 (classification1 (cocone-diagram ?s)))
              (= (CLS.INFO$target (opfirst ?s)) (opvertex ?s))))

(11) (KIF$function opsecond)
     (= (KIF$source opsecond) cocone)
     (= (KIF$target opsecond) CLS.INFO$infomorphism)
     (forall (?s (cocone ?s))
         (and (= (CLS.INFO$source (opsecond ?s))
                 (classification2 (cocone-diagram ?s)))
              (= (CLS.INFO$target (opsecond ?s)) (opvertex ?s))))
```

- There is an *instance cone* function, which maps a binary coproduct cocone of classifications and infomorphisms to the underlying binary product cone of instance classes and instance functions. Similarly, there is a *type cocone* function, which maps a binary coproduct cocone of classifications and infomorphisms to the underlying binary coproduct cocone of type classes and type functions.

```
(12) (KIF$function instance-cone)
     (= (KIF$source instance-cone) cocone)
     (= (KIF$target instance-cone) SET.LIM.PRD$cone)
     (forall (?s (cocone ?s))
         (and (= (SET.LIM.PRD$cone-diagram (instance-cone ?s))
                 (instance-diagram (cocone-diagram ?s)))
              (= (SET.LIM.PRD$vertex (instance-cone ?s))
                 (CLS$instance (opvertex ?s)))
              (= (SET.LIM.PRD$first (instance-pair ?s))
                 (CLS.INFO$instance (opfirst ?s)))
              (= (SET.LIM.PRD$second (instance-pair ?s))
                 (CLS.INFO$instance (opsecond ?s)))))

(13) (KIF$function type-cocone)
     (= (KIF$source type-cocone) cocone)
     (= (KIF$target type-cocone) SET.COLIM.COPRD$cocone)
     (forall (?s (cocone ?s))
         (and (= (SET.COLIM.COPRD$cocone-diagram (type-cocone ?s))
                 (type-diagram (cocone-diagram ?s)))
              (= (SET.COLIM.COPRD$opvertex (type-cocone ?s))
                 (CLS$type (opvertex ?s)))
              (= (SET.COLIM.COPRD$opfirst (type-cocone ?s))
                 (CLS.INFO$type (opfirst ?s)))
              (= (SET.COLIM.COPRD$opsecond (type-cocone ?s))
                 (CLS.INFO$type (opsecond ?s)))))
```

○ There is a KIF function '`colimiting-cone`' that maps a pair (of classifications) to its binary coproduct (colimiting binary coproduct cocone) (see Figure 21, where arrows denote functions). The totality of this function, along with the universality of the comediator infomorphism, implies that a binary coproduct exists for any pair of classifications. The opvertex of the colimiting binary coproduct cocone is a specific *binary coproduct* class. It comes equipped with two *injection* infomorphisms. The binary coproduct and injections are expressed both abstractly in the second to last axiom and concretely in the last axiom. The last axiom implicitly ensures that both the coproduct and the two injection infomorphisms are specific – that its instance class is exactly the Cartesian product of the instance classes of the pair of classifications and that its type class is exactly the disjoint union of the type classes of the pair of classifications.

**Figure 21: Colimiting Cocone**

$$\begin{tikzcd}
\boldsymbol{A}_1 \arrow[dr, shift left] & & \boldsymbol{A}_2 \arrow[dl, shift right] \\
& \boldsymbol{A}_1+\boldsymbol{A}_2 \arrow[ul, shift left, "\iota_1"] \arrow[ur, shift right, "\iota_2"'] &
\end{tikzcd}$$

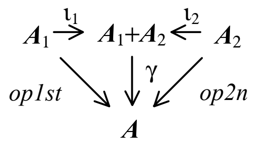
```
(14) (KIF$function colimiting-cocone)
     (= (KIF$source colimiting-cocone) diagram)
     (= (KIF$target colimiting-cocone) cocone)
     (forall (?p (diagram ?p))
         (= (cocone-diagram (colimiting-cocone ?p)) ?p))

(15) (KIF$function colimit)
     (KIF$function binary-coproduct)
     (= binary-coproduct colimit)
     (= (KIF$source colimit) diagram)
     (= (KIF$target colimit) CLS$classification)

(16) (KIF$function injection1)
     (= (KIF$source injection1) diagram)
     (= (KIF$target injection1) CLS.INFO$infomorphism)

(17) (KIF$function injection2)
     (= (KIF$source injection2) diagram)
     (= (KIF$target injection2) CLS.INFO$infomorphism)

(18) (forall (?p (diagram ?p))
         (and (= (colimit ?p) (opvertex (colimiting-cocone ?p)))
              (= (CLS.INFO$source (injection1 ?p)) (classification1 ?p))
              (= (CLS.INFO$target (injection1 ?p)) (colimit ?p))
              (= (injection1 ?p) (opfirst (colimiting-cocone ?p)))
              (= (CLS.INFO$source (injection2 ?p)) (classification2 ?p))
              (= (CLS.INFO$target (injection2 ?p)) (colimit ?p))
              (= (injection2 ?p) (opsecond (colimiting-cocone ?p)))))

(19) (forall (?p (diagram ?p)
              ?i ((CLS$instance (colimit ?p)) ?i)
              ?t ((CLS$type (colimit ?p)) ?t))
         (<=> ((colimit ?p) ?i ?t)
              (and (=> (= (?t 1) 1) ((classification1 ?p) (?i 1) (?t 2)))
                   (=> (= (?t 1) 2) ((classification2 ?p) (?i 2) (?t 2))))))
```

o The following two axioms are the necessary conditions that the instance and type quasi-functors preserve concrete colimits. These explicitly ensure that this colimit is specific – that its instance class is exactly the Cartesian product of the instance classes of the pair of classifications and that its type class is exactly the disjoint union of the type classes of the pair of classifications. Also, these explicitly ensure that the two coproduct injection infomorphisms are specific – that their instance functions are exactly the Cartesian product projections of the instance classes of the pair of classifications and that their type functions are exactly the disjoint union injections of the type classes of the pair of classifications.

```
(20) (forall (?p (diagram ?p))
         (and (= (instance-cone (colimiting-cocone ?p))
                 (SET.LIM.PRD$limiting-cone (instance-diagram ?p)))
              (= (CLS$instance (colimit ?p))
                 (SET.LIM.PRD$limit (instance-diagram ?p)))
              (= (CLS.INFO$instance (injection1 ?p))
                 (SET.LIM.PRD$projection1 (instance-diagram ?p)))
              (= (CLS.INFO$instance (injection2 ?p))
                 (SET.LIM.PRD$projection2 (instance-diagram ?p)))))

(21) (forall (?p (diagram ?p))
         (and (= (type-cocone (colimiting-cocone ?p))
                 (SET.COL.COPRD$colimiting-cocone (type-diagram ?p)))
              (= (CLS$type (colimit ?p))
                 (SET.COL.COPRD$colimit (type-diagram ?p)))
              (= (CLS.INFO$type (injection1 ?p))
                 (SET.COL.COPRD$injection1 (type-diagram ?p)))
              (= (CLS.INFO$type (injection2 ?p))
                 (SET.COL.COPRD$injection2 (type-diagram ?p)))))
```

- For any binary coproduct cocone, there is a *comediator* infomorphism $\gamma : A_1+A_2 \rightleftarrows A$ (see Figure 22, where arrows denote infomorphisms) from the binary coproduct of the underlying diagram (pair of classifications) to the opvertex of the cocone. This is the unique infomorphism, which commutes with opfirst and opsecond. We define this by using the mediator of the underlying instance cone and the comediator of the underlying type cocone. Existence and uniqueness represents the universality of the binary coproduct operator.

**Figure 22: Coproduct Comediator**


$$\begin{tikzcd}
A_1 \arrow[r, "\iota_1"] \arrow[dr, "op1st"'] & A_1+A_2 \arrow[d, "\gamma"] & A_2 \arrow[l, "\iota_2"'] \arrow[dl, "op2n"] \\
& A &
\end{tikzcd}$$

```
(22) (KIF$function comediator)
     (= (KIF$source comediator) cocone)
     (= (KIF$target comediator) CLS.INFO$infomorphism)
     (forall (?s (cocone ?s))
         (and (= (CLS.INFO$source (comediator ?s)) (colimit (cocone-diagram ?s)))
              (= (CLS.INFO$target (comediator ?s)) (opvertex ?s))
              (= (CLS.INFO$instance (comediator ?s))
                 (SET.LIM.PRD$mediator (instance-cone ?s)))
              (= (CLS.INFO$type (comediator ?s))
                 (SET.COL.COPRD$comediator (type-cocone ?s)))))
```

- It can be verified that the comediator is the unique infomorphism that makes the diagram in Figure 6 commutative.

```
     (forall (?s (cocone ?s))
         (= (comediator ?s)
            (the (?f (CLS.INFO$infomorphism ?f))
                (and (= (CLS.INFO$composition [(injection1 (cocone-diagram ?s)) ?f])
                        (opfirst ?s))
                     (= (CLS.INFO$composition [(injection2 (cocone-diagram ?s)) ?f])
                        (opsecond ?s))))))
```

## Coinvariants and Coquotients

**`CLS.COL.COINV`**

- Given a classification $\boldsymbol{A} = \langle inst(\boldsymbol{A}), typ(\boldsymbol{A}), \vDash_{\boldsymbol{A}} \rangle$, a *coinvariant* (called a *dual invariant* in Barwise and Seligman, 1997) is a pair $J = \langle A, R \rangle$ consisting of a sub*class* of instances $A \subseteq inst(\boldsymbol{A})$ and a binary *endorelation* $\boldsymbol{R}$ on types $R \subseteq typ(\boldsymbol{A}) \times typ(\boldsymbol{A})$ that satisfies the *fundamental constraint*:

  if $(t_0, t_1) \in R$ then for each $i \in A$, $i \vDash_{\boldsymbol{A}} t_0$ iff $i \vDash_{\boldsymbol{A}} t_1$.

  The classification $\boldsymbol{A}$ is called the *base classification* of $J$ – the classification on which $J$ is based. A coinvariant is determined by its base, class and endorelation triple.

```
(1) (KIF$collection coinvariant)

(2) (KIF$function classification)
    (KIF$function base)
    (= base classification)
```

```
        (= (KIF$source base) coinvariant)
        (= (KIF$target base) CLS$classification)

    (3) (KIF$function class)
        (= (KIF$source class) coinvariant)
        (= (KIF$target class) SET$class)

    (4) (KIF$function endorelation)
        (= (KIF$source endorelation) coinvariant)
        (= (KIF$target endorelation) REL.ENDO$endorelation)

    (5) (forall (?j (coinvariant ?j))
            (and (SET$subclass (class ?j) (CLS$instance (base ?j)))
                 (= (REL.ENDO$class (endorelation ?j)) (CLS$type (base ?j)))
                 (forall (?t0 ?t1 ((endorelation ?j) ?t0 ?t1)
                          ?i ((class ?j) ?i))
                     (<=> ((base ?j) ?i ?t0)
                          ((base ?j) ?i ?t1)))))

        (forall (?j1 (coinvariant ?j1) ?j2 (coinvariant ?j2))
            (=> (and (= (base ?j1) (base ?j2))
                     (= (class ?j1) (class ?j2))
                     (= (endorelation ?j1) (endorelation ?j2)))
                (= ?j1 ?j2)))
```

○ Often, the relation $\boldsymbol{R}$ is an equivalence relation on the types. However (Barwise and Seligman, 1997), it is convenient not to require this. The endorelation $\boldsymbol{R}$ is contained in a smallest equivalence relation $\equiv_{\boldsymbol{R}}$ on types called the equivalence relation generated by $\boldsymbol{R}$. This is the reflexive, symmetric, transitive closure of $\boldsymbol{R}$. For any type $t \in \mathsf{typ}(\boldsymbol{A})$, write $[t]_{\boldsymbol{R}}$ for the $\boldsymbol{R}$-equivalence class of $t$. Then $\hat{J} = \langle A, \equiv_{\boldsymbol{R}} \rangle$ is also a coinvariant on $\boldsymbol{A}$.

The *coquotient* $\boldsymbol{A}/J$ of a coinvariant $J$ on a classification $\boldsymbol{A}$ (see Figure 23, where arrows denote functions) (called the *dual quotient* in Barwise and Seligman, 1997) is the classification defined as follows:

– The class of instances of $\boldsymbol{A}/J$ is $A$, the given subset of $\mathsf{inst}(\boldsymbol{A})$.

– The class of types of $\boldsymbol{A}/J$ is $\mathsf{typ}(\boldsymbol{A})/\boldsymbol{R}$, the quotient class over $\mathsf{typ}(\boldsymbol{A})$ of $\boldsymbol{R}$-equivalence classes.

– The incidence $\vDash_{\boldsymbol{A}/J}$ of $\boldsymbol{A}/J$ is defined by

$$i \vDash_{A/J} [t]_R \text{ when } i \vDash_{\boldsymbol{A}} t$$

which is well-defined by the fundamental constraint.

There is a *canonical quotient infomorphism* $\boldsymbol{\tau}_J : \boldsymbol{A} \rightleftarrows \boldsymbol{A}/J$, whose instance function is the inclusion function $inc : A \to \mathsf{inst}(\boldsymbol{A})$, and whose type function is the canonical quotient function $[\text{-}]_{\boldsymbol{R}} : \mathsf{typ}(\boldsymbol{A}) \to \mathsf{typ}(\boldsymbol{A})/\boldsymbol{R}$. The fundamental property for this infomorphism is trivial, given the definition of the coquotient incidence above.

$$\begin{tikzcd}
\boldsymbol{A} \arrow[r, shift left, "\boldsymbol{\tau}_J"] \arrow[dr, shift left] & \boldsymbol{A}/J \arrow[l, shift left] \arrow[d, shift left, "\tilde{f}"] \\
 & \boldsymbol{B} \arrow[ul, shift left, "f"] \arrow[u, shift left]
\end{tikzcd}$$

**Figure 23: Universality of the Coquotient**

```
    (6) (KIF$function coquotient)
        (= (KIF$source coquotient) coinvariant)
        (= (KIF$target coquotient) CLS$classification)
        (forall (?j (coinvariant ?j))
            (and (= (CLS$instance (coquotient ?j)) (class ?j))
                 (= (CLS$type (coquotient ?j))
                    (REL.ENDO$quotient (REL.ENDO$equivalence-closure (endorelation ?j))))
                 (forall (?i ((class ?j) ?i)
                          ?t ((CLS$type (base ?j)) ?t))
                     (<=> ((coquotient ?j)
                           ?i
                           ((REL.ENDO$canon
                               (REL.ENDO$equivalence-closure (endorelation ?j))) ?t))
                          ((base ?j) ?i ?t)))))

    (7) (KIF$function canon)
        (= (KIF$source canon) coinvariant)
        (= (KIF$target canon) CLS.INFO$infomorphism)
        (forall (?j (coinvariant ?j))
            (and (= (CLS.INFO$source (canon ?j)) (base ?j))
                 (= (CLS.INFO$target (canon ?j)) (coquotient ?j))
                 (= (CLS.INFO$instance (canon ?j))
                    (SET.FTN$inclusion [(class ?j) (CLS$instance (base ?j))]))
                 (= (CLS.INFO$type (canon ?j))
                    (REL.ENDO$canon (REL.ENDO$equivalence-closure (endorelation ?j))))))
```

- Let $J = \langle A, R \rangle$ be a coinvariant on a classification $\boldsymbol{A}$. An infomorphism $\boldsymbol{f} : \boldsymbol{A} \rightleftarrows \boldsymbol{B}$ *respects* $J$ when:
  - for any instance $j \in inst(\boldsymbol{B})$, $inst(\boldsymbol{f})(j) \in A$; and
  - for any two types $t_0, t_1 \in typ(\boldsymbol{A})$, if $(t_0, t_1) \in R$ then $typ(\boldsymbol{f})(t_0) = typ(\boldsymbol{f})(t_1)$.

```
(8) (KIF$relation respects)
    (= (KIF$collection1 respects) CLS.INFO$infomorphism)
    (= (KIF$collection2 respects) coinvariant)
    (forall (?f (CLS.INFO$infomorphism ?f)
             ?J (coinvariant ?J))
        (<=> (respects ?f ?J)
             (and (= (CLS.INFO$source ?f) (base ?J))
                  (forall (?j ((CLS$instance (CLS.INFO$target ?f)) ?j))
                      ((class ?J) ((CLS.INFO$instance ?f) ?j)))
                  (forall (?t0 ((CLS$type (CLS.INFO$source ?f)) ?t0)
                           ?t1 ((CLS$type (CLS.INFO$source ?f)) ?t1)
                           ((endorelation ?j) ?t0 ?t1))
                      (= ((CLS.INFO$type ?f) ?t0) ((CLS.INFO$type ?f) ?t1))))))
```

**Proposition.** *For every coinvariant J on a classification* $\boldsymbol{A}$ *and every infomorphism* $\boldsymbol{f} : \boldsymbol{A} \rightleftarrows \boldsymbol{B}$ *that respects J, there is a unique comediating infomorphism* $\tilde{\boldsymbol{f}} : \boldsymbol{A}/J \rightleftarrows \boldsymbol{B}$ *such that* $\boldsymbol{\tau_J} \circ \tilde{\boldsymbol{f}} = \boldsymbol{f}$ *(the diagram in Figure 7 commutes).*

Based on this proposition, a definite description is used to define a *comediator* function (Figure 7) that maps a pair $(\boldsymbol{f}, J)$ consisting of a coinvariant and a respectful infomorphism to their comediator $\tilde{\boldsymbol{f}}$.

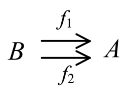
```
(9) (KIF$function comediator)
    (= (KIF$source comediator) (KIF$extent respects))
    (= (KIF$target comediator) CLS.INFO$infomorphism)
    (forall (?j (coinvariant ?j)
             ?f (CLS.INFO$infomorphism ?f) (respects ?f ?j))
        (= (comediator [?f ?j])
           (the (?ft (CLS.INFO$infomorphism ?ft))
               (and (= (CLS.INFO$source ?ft) (coquotient ?j))
                    (= (CLS.INFO$target ?ft) (CLS.INFO$target ?f))
                    (= (CLS.INFO$composition [(canon ?j) ?ft]) ?f)))))
```

## Coequalizers

**`CLS.COL.COEQ`**

A *coequalizer* is a finite colimit in CLASSIFICATION for a diagram of shape *parallel-pair* = $\cdot \rightrightarrows \cdot$. Such a diagram (of classes and functions) is called a *parallel pair* of functions.

- A *parallel pair* (see Figure 24, where arrows denote infomorphisms) is the appropriate base diagram for a coequalizer. Each parallel pair consists of a pair of infomorphisms called *infomorphism1* and *infomorphism2* that share the same *source* and *target* classifications. We use either the generic term '`diagram`' or the specific term '`parallel-pair`' to denote the *parallel pair* collection. Parallel pairs are determined by their two component infomorphisms.



$$\begin{tikzcd} B \arrow[r, shift left, "f_1"] \arrow[r, shift right, "f_2"'] & A \end{tikzcd}$$

**Figure 24: Parallel Pair**

```
(1) (KIF$collection diagram)
    (KIF$collection parallel-pair)
    (= parallel-pair diagram)

(2) (KIF$function source)
    (= (KIF$source source) diagram)
    (= (KIF$target source) CLS$classification)

(3) (KIF$function target)
```

```
        (= (KIF$source target) diagram)
        (= (KIF$target target) CLS$classification)

    (4) (KIF$function infomorphism1)
        (= (KIF$source infomorphism1) diagram)
        (= (KIF$target infomorphism1) CLS.INFO$infomorphism)

    (5) (KIF$function infomorphism2)
        (= (KIF$source infomorphism2) diagram)
        (= (KIF$target infomorphism2) CLS.INFO$infomorphism)

    (6) (forall (?p (diagram ?p))
            (and (= (SET.FTN$source (infomorphism1 ?p)) (source ?p))
                 (= (SET.FTN$target (infomorphism1 ?p)) (target ?p))
                 (= (SET.FTN$source (infomorphism2 ?p)) (source ?p))
                 (= (SET.FTN$target (infomorphism2 ?p)) (target ?p))))

        (forall (?p (diagram ?p) ?q (diagram ?q))
            (=> (and (= (infomorphism1 ?p) (infomorphism1 ?q))
                     (= (infomorphism2 ?p) (infomorphism2 ?q)))
                (= ?p ?q)))
```

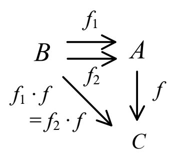
- There is an *instance parallel pair* or *instance diagram* function, which maps a parallel pair of infomorphisms to the underlying (SET.LIM.EQU) parallel pair of instance functions. Similarly, there is a *type parallel pair* or *type diagram* function, which maps a parallel pair of infomorphisms to the underlying (SET.COLIM.COEQ) parallel pair of type functions.

```
    (7) (KIF$function instance-diagram)
        (KIF$function instance-parallel-pair)
        (= instance-parallel-pair instance-diagram)
        (= (KIF$source instance-diagram) diagram)
        (= (KIF$target instance-diagram) SET.LIM.EQU$diagram)
        (forall (?p (diagram ?p))
            (and (= (SET.LIM.EQU$source (instance-diagram ?p))
                    (CLS$instance (target ?p)))
                 (= (SET.LIM.EQU$target (instance-diagram ?p))
                    (CLS$instance (source ?p)))
                 (= (SET.LIM.EQU$function1 (instance-diagram ?p))
                    (CLS.INFO$instance (infomorphism1 ?p)))
                 (= (SET.LIM.EQU$function2 (instance-diagram ?p))
                    (CLS.INFO$instance (infomorphism2 ?p)))))

    (8) (KIF$function type-diagram)
        (KIF$function type-parallel-pair)
        (= type-parallel-pair type-diagram)
        (= (KIF$source type-diagram) diagram)
        (= (KIF$target type-diagram) SET.COLIM.COEQ$diagram)
        (forall (?p (diagram ?p))
            (and (= (SET.COLIM.COEQ$source (type-diagram ?p))
                    (CLS$type (source ?p)))
                 (= (SET.COLIM.COEQ$target (type-diagram ?p))
                    (CLS$type (target ?p)))
                 (= (SET.COLIM.COEQ$function1 (type-diagram ?p))
                    (CLS.INFO$type (infomorphism1 ?p)))
                 (= (SET.COLIM.COEQ$function2 (type-diagram ?p))
                    (CLS.INFO$type (infomorphism2 ?p)))))
```

- The information in a coequalizer diagram (parallel pair of infomorphisms) is equivalently expressed as a *coinvariant* based on the target classification of the parallel pair, whose class is the equalizer of instance diagram, and whose endorelation is the coequalizer endorelation of the type diagram.

```
    (9) (KIF$function coinvariant)
        (= (KIF$source coinvariant) diagram)
        (= (KIF$target coinvariant) CLS.COL.COINV$coinvariant)
        (forall (?p (diagram ?p))
            (and (= (CLS.COL.COINV$base (coinvariant ?p)) (target ?p))
                 (= (CLS.COL.COINV$class (coinvariant ?p))
                    (SET.LIM.EQU$equalizer (instance-diagram ?p)))
                 (= (CLS.COL.COINV$endorelation (coinvariant ?p))
                    (SET.COL.COEQ$endorelation (type-diagram ?p)))))
```

- The notion of a *coequalizer cocone* is used to specify and axiomatize coequalizers. Each coequalizer cocone (see Figure 25, where arrows denote infomorphisms) has an *opvertex* classification, and an *infomorphism* whose source classification is the target classification of the infomorphisms in the parallel-pair and whose target classification is the opvertex. Since Figure 25 is commutative, the composite infomorphism is not needed. Each coequalizer cocone is situated under a coequalizer *diagram* (parallel pair of infomorphisms).



$$\begin{tikzcd} B \arrow[r, shift left, "f_1"] \arrow[r, shift right, "f_2"'] \arrow[dr, "f_1 \cdot f = f_2 \cdot f"'] & A \arrow[d, "f"] \\ & C \end{tikzcd}$$

**Figure 25: Coequalizer Cocone**

```
(10) (KIF$collection cocone)

(11) (KIF$function cocone-diagram)
     (= (KIF$source cocone-diagram) cocone)
     (= (KIF$target cocone-diagram) diagram)

(12) (KIF$function opvertex)
     (= (KIF$source opvertex) cocone)
     (= (KIF$target opvertex) CLS$classification)

(13) (KIF$function infomorphism)
     (= (KIF$source infomorphism) cocone)
     (= (KIF$target infomorphism) CLS.INFO$infomorphism)

(14) (forall (?s (cocone ?s))
        (and (= (CLS.INFO$source (infomorphism ?s)) (target (cocone-diagram ?s)))
             (= (CLS.INFO$target (infomorphism ?s)) (opvertex ?s))
             (= (CLS.INFO$composition
                    [(infomorphism1 (cocone-diagram ?s))
                     (infomorphism ?s)])
                (CLS.INFO$composition
                    [(infomorphism2 (cocone-diagram ?s))
                     (infomorphism ?s)]))))
```

- The *instance equalizer cone* function maps a coequalizer cocone of classifications and infomorphisms to the underlying equalizer cone of instance classes and instance functions. Dually, the *type coequalizer cocone* function maps a coequalizer cocone of classifications and infomorphisms to the underlying coequalizer cocone of type classes and type functions.

```
(15) (KIF$function instance-cone)
     (= (KIF$source instance-cone) cocone)
     (= (KIF$target instance-cone) SET.LIM.EQU$cone)
     (forall (?s (cocone ?s))
         (and (= (SET.LIM.EQU$cone-diagram (instance-cone ?s))
                 (instance-diagram (cocone-diagram ?s)))
              (= (SET.LIM.EQU$vertex (instance-cone ?s))
                 (CLS$instance (opvertex ?s)))
              (= (SET.LIM.EQU$function (instance-pair ?s))
                 (CLS.INFO$instance (infomorphism ?s)))))

(16) (KIF$function type-cocone)
     (= (KIF$source type-cocone) cocone)
     (= (KIF$target type-cocone) SET.COLIM.COEQ$cocone)
     (forall (?s (cocone ?s))
         (and (= (SET.COLIM.COEQ$cocone-diagram (type-cocone ?s))
                 (type-diagram (cocone-diagram ?s)))
              (= (SET.COLIM.COEQ$opvertex (type-cocone ?s))
                 (CLS$type (opvertex ?s)))
              (= (SET.COLIM.COEQ$function (type-cocone ?s))
                 (CLS.INFO$type (infomorphism ?s)))))
```

- It is important to observe that the infomorphism in a cocone respects the coinvariant of the underlying diagram of the cocone. This fact is needed to help define the comediator of a cocone.

```
     (forall (?s (cocone ?s))
         (CLS.COL.COINV$respects (infomorphism ?s) (coinvariant (cocone-diagram ?s))))
```

- There is a KIF function ‘`limiting-cone`’ that maps a parallel pair (of infomorphisms) to its coequalizer (colimiting coequalizer cocone) (see Figure 26, where arrows denote infomorphisms). The totality of this function, along with the universality of the comediator infomorphism, implies that a coequalizer exists for any parallel pair of infomorphisms. The opvertex of the colimiting coequalizer cocone is a specific *coequalizer* class. It comes equipped with a *canon* infomorphism. The coequalizer and canon are expressed both abstractly, and, in the last axiom, as the coquotient and canon of the coinvariant of the diagram.

**Figure 26: Colimiting Cocone**

$$\begin{tikzcd}
\boldsymbol{B} \arrow[r, shift left, "f_1"] \arrow[r, shift right, "f_2"'] \arrow[dr] & \boldsymbol{A} \arrow[d, "\gamma"] \\
 & \hat{\boldsymbol{A}}
\end{tikzcd}$$

```
(17) (KIF$function colimiting-cocone)
     (= (KIF$source colimiting-cocone) diagram)
     (= (KIF$target colimiting-cocone) cocone)
     (forall (?p (diagram ?p))
         (= (cocone-diagram (colimiting-cocone ?p)) ?p))

(18) (KIF$function colimit)
     (KIF$function coequalizer)
     (= coequalizer colimit)
     (= (KIF$source colimit) diagram)
     (= (KIF$target colimit) CLS$classification)

(19) (KIF$function canon)
     (= (KIF$source canon) diagram)
     (= (KIF$target canon) CLS.INFO$infomorphism)

(20) (forall (?p (diagram ?p))
         (and (= (colimit ?p) (opvertex (colimiting-cocone ?p)))
              (= (CLS.INFO$source (canon ?p)) (target ?p))
              (= (CLS.INFO$target (canon ?p)) (colimit ?p))
              (= (canon ?p) (infomorphism (colimiting-cocone ?p)))))

(21) (forall (?p (diagram ?p))
        (and (= (colimit ?p) (CLS.COL.COINV$coquotient (coinvariant ?p)))
             (= (canon ?p) (CLS.COL.COINV$canon (coinvariant ?p)))))
```

- The following two axioms are the necessary conditions that the instance and type quasi-functors preserve concrete colimits. These ensure that both this coequalizer and its canon infomorphism are specific.

```
(22) (forall (?p (diagram ?p))
         (and (= (instance-cone (colimiting-cocone ?p))
                 (SET.LIM.EQU$limiting-cone (instance-diagram ?p)))
              (= (CLS$instance (colimit ?p))
                 (SET.LIM.EQU$limit (instance-diagram ?p)))
              (= (CLS.INFO$instance (canon ?p))
                 (SET.LIM.EQU$inclusion (instance-diagram ?p)))))

(23) (forall (?p (diagram ?p))
         (and (= (type-cocone (colimiting-cocone ?p))
                 (SET.COL.COEQ$colimiting-cocone (type-diagram ?p)))
              (= (CLS$type (colimit ?p))
                 (SET.COL.COEQ$colimit (type-diagram ?p)))
              (= (CLS.INFO$type (canon ?p))
                 (SET.COL.COEQ$canon (type-diagram ?p)))))
```

- The *comediator* infomorphism, from the coequalizer of a parallel pair of infomorphisms to the opvertex of a cocone under the parallel pair (see Figure 27, where arrows denote infomorphisms), is the unique infomorphism that commutes with cocone infomorphisms. This is defined abstractly by using a definite description, and is defined concretely as the comediator of the associated coinvariant.

**Figure 27: Comediator**

$$\begin{tikzcd}
\boldsymbol{B} \arrow[r, shift left, "f_1"] \arrow[r, shift right, "f_2"'] & \boldsymbol{A} \arrow[r, "\gamma"] \arrow[dr, "g"'] & \hat{\boldsymbol{A}} \arrow[d, "\hat{g}"] \\
 & & \boldsymbol{C}
\end{tikzcd}$$

```
(24) (KIF$function comediator)
     (= (KIF$source comediator) cocone)
     (= (KIF$target comediator) CLS.INFO$infomorphism)
     (forall (?s (cocone ?s))
         (and (= (CLS.INFO$source (comediator ?s)) (colimit (cocone-diagram ?s)))
              (= (CLS.INFO$target (comediator ?s)) (opvertex ?s))
              (= (comediator ?s)
                 (the (?m (CLS.INFO$infomorphism ?m))
                     (= (composition [(canon (cocone-diagram ?s)) ?m])
                        (infomorphism ?s))))))

(25) (forall (?s (cocone ?s))
         (= (comediator ?s)
            (CLS.COL.COINV$comediator
                [(infomorphism ?s) (coinvariant (cocone-diagram ?s))])))
```

## Pushouts

**CLS.COL.PSH**

A *pushout* is a finite colimit for a diagram of shape *span* = • ← • → •. Such a diagram (of classifications and infomorphisms) is called an *span* (see Figure 28, where arrows denote infomorphisms)

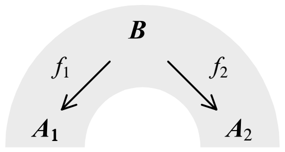


**Figure 28: Pushout Diagram = Span**

- A *span* is the appropriate base diagram for a pushout. Each opspan consists of a pair of infomorphisms called *first* and *second*. These are required to have a common source classification ***B***, denoted as the *vertex*. We use either the generic term '`diagram`' or the specific term '`span`' to denote the *span* collection. A span is the special case of a general diagram whose shape is the graph that is also named *span*. Spans are determined by their pair of component infomorphisms.

```
    (1) (KIF$collection diagram)
        (KIF$collection span)
        (= span diagram)

    (2) (KIF$function classification1)
        (= (KIF$source classification1) diagram)
        (= (KIF$target classification1) CLS$classification)

    (3) (KIF$function classification2)
        (= (KIF$source classification2) diagram)
        (= (KIF$target classification2) CLS$classification)

    (4) (KIF$function vertex)
        (= (KIF$source vertex) diagram)
        (= (KIF$target vertex) CLS$classification)

    (5) (KIF$function first)
        (= (KIF$source first) diagram)
        (= (KIF$target first) CLS.INFO$infomorphism)

    (6) (KIF$function second)
        (= (KIF$source second) diagram)
        (= (KIF$target second) CLS.INFO$infomorphism)

    (7) (forall (?r (diagram ?r))
           (and (= (CLS.INFO$source (first ?r)) (vertex ?r))
                (= (CLS.INFO$source (second ?r)) (vertex ?r))
                (= (CLS.INFO$target (first ?r)) (classification1 ?r))
                (= (CLS.INFO$target (second ?r)) (classification2 ?r))))

        (forall (?r1 (diagram ?r1) ?r2 (diagram ?r2))
           (=> (and (= (first ?r1) (first ?r2))
                    (= (second ?r1) (second ?r2)))
               (= ?r1 ?r2)))
```

- The *pair* of target classifications (suffixing discrete diagram) underlying any span is named. This construction is derived from the fact that the pair shape is a subshape of span shape.

```
    (8) (KIF$function pair)
        (= (KIF$source pair) diagram)
        (= (KIF$target pair) CLS.COL.COPRD$diagram)
        (forall (?r (diagram ?r))
            (and (CLS.COL.COPRD$classification1 (pair ?r)) (classification1 ?r))
                 (CLS.COL.COPRD$classification2 (pair ?r)) (classification2 ?r))))
```

o Every span has an opposite.

```
(9) (KIF$function opposite)
    (= (KIF$source opposite) span)
    (= (KIF$target opposite) span)
    (forall (?r (span ?r))
       (and (= (classification1 (opposite ?r)) (classification2 ?r))
            (= (classification2 (opposite ?r)) (classification1 ?r))
            (= (vertex (opposite ?r)) (vertex ?r))
            (= (first (opposite ?r)) (second ?r))
            (= (second (opposite ?r)) (first ?r))))
```

o The opposite of the opposite is the original opspan – the following theorem can be proven.

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

o The *instance opspan* or *instance diagram* function maps a span of infomorphisms to the underlying (`SET.LIM.PBK`) opspan of instance functions. Dually, the *type span* or *type diagram* function maps a span of infomorphisms to the underlying (`SET.COLIM.COEQ`) span of type functions.

```
(10) (KIF$function instance-diagram)
     (KIF$function instance-opspan)
     (= instance-opspan instance-diagram)
     (= (KIF$source instance-diagram) diagram)
     (= (KIF$target instance-diagram) SET.LIM.PBK$diagram)
     (forall (?r (diagram ?r))
         (and (= (SET.LIM.PBK$class1 (instance-diagram ?r))
                 (CLS$instance (classification1 ?r)))
              (= (SET.LIM.PBK$class2 (instance-diagram ?r))
                 (CLS$instance (classification2 ?r)))
              (= (SET.LIM.PBK$opvertex (instance-diagram ?r))
                 (CLS$instance (vertex ?r)))
              (= (SET.LIM.PBK$opfirst (instance-diagram ?r))
                 (CLS.INFO$instance (first ?r)))
              (= (SET.LIM.PBK$opsecond (instance-diagram ?r))
                 (CLS.INFO$instance (second ?r)))))

(11) (KIF$function type-diagram)
     (KIF$function type-span)
     (= type-span type-diagram)
     (= (KIF$source type-diagram) diagram)
     (= (KIF$target type-diagram) SET.COL.PSH$diagram)
     (forall (?r (diagram ?r))
         (and (= (SET.COL.PSH$class1 (type-diagram ?r))
                 (CLS$type (classification1 ?r)))
              (= (SET.COL.PSH$class2 (type-diagram ?r))
                 (CLS$type (classification2 ?r)))
              (= (SET.COL.PSH$vertex (type-diagram ?r))
                 (CLS$type (vertex ?r)))
              (= (SET.COL.PSH$first (type-diagram ?r))
                 (CLS.INFO$type (first ?r)))
              (= (SET.COL.PSH$second (type-diagram ?r))
                 (CLS.INFO$type (second ?r)))))
```

o The *parallel pair* or *coequalizer diagram* function maps a span of infomorphisms to the associated (`CLS.COL.COEQ`) parallel pair of infomorphisms (see Figure 29, where arrows denote infomorphisms), which are the composite of the first and second infomorphisms of the span with the coproduct injection infomorphisms of the binary coproduct of the component classifications in the span. The coequalizer and canon of the associated parallel pair will be used to define the pushout.

$$\begin{tikzcd}
 & \boldsymbol{B} \arrow[dl, "f_1"'] \arrow[d, shift left] \arrow[d, shift right] \arrow[dr, "f_2"] & \\
\boldsymbol{A}_1 \arrow[r, "\iota_1"'] & \boldsymbol{A}_1+\boldsymbol{A}_2 & \boldsymbol{A}_2 \arrow[l, "\iota_2"]
\end{tikzcd}$$

**Figure 29: Coequalizer Diagram**

```
(12) (KIF$function coequalizer-diagram)
     (KIF$function parallel-pair)
```

```
        (= parallel-pair coequalizer-diagram)
        (= (KIF$source coequalizer-diagram) diagram)
        (= (KIF$target coequalizer-diagram) CLS.COL.COEQ$diagram)
        (forall (?r (diagram ?r))
            (and (= (CLS.COL.COEQ$source (coequalizer-diagram ?r))
                    (vertex ?r))
                 (= (CLS.COL.COEQ$target (coequalizer-diagram ?r))
                    (CLS.COL.COPRD$binary-coproduct (pair ?r)))
                 (= (CLS.COL.COEQ$infomorphism1 (coequalizer-diagram ?r))
                    (CLS.INFO$composition
                        [(first ?r)
                         (CLS.COL.COPRD$injection1 (pair ?r))]))
                 (= (CLS.COL.COEQ$infomorphism2 (coequalizer-diagram ?r))
                    (CLS.INFO$composition
                        [(second ?r)
                         (CLS.COL.COPRD$injection2 (pair ?r))]))))
```

- There are two important categorical identities (not just isomorphisms) that relate (1) the underlying instance limit and (2) the underlying type colimit of the coequalizer of the parallel pair of any span to (1′) the limit (equalizer) of the equalizer diagram of the underlying instance opspan and (2′) the colimit (coequalizer) of the coequalizer diagram of the underlying type span. These identities are assumed in the definition of the colimiting cocone below.

```
(13) (forall (?r (diagram ?r))
         (and (= (CLS.COL.COEQ$instance-diagram (coequalizer-diagram ?r))
                 (SET.LIM.EQU$equalizer-diagram (instance-diagram ?r)))
              (= (CLS.COL.COEQ$type-diagram (coequalizer-diagram ?r))
                 (SET.COL.COEQ$coequalizer-diagram (type-diagram ?r)))))

(14) (forall (?r (diagram ?r))
         (and (= (CLS.COL.COEQ$instance-diagram
                     (CLS.COL.COEQ$colimiting-cocone (coequalizer-diagram ?r)))
                 (SET.LIM.EQU$limiting-cone
                     (SET.LIM.EQU$equalizer-diagram (instance-diagram ?r))))
              (= (CLS.COL.COEQ$type-diagram
                     (CLS.COL.COEQ$colimiting-cocone (coequalizer-diagram ?r)))
                 (SET.LIM.COEQ$colimiting-cocone
                     (SET.COL.COEQ$coequalizer-diagram (type-diagram ?r))))))
```

- *Pushout cocones* are used to specify and axiomatize pushouts. Each pushout cocone (Figure 30, where arrows denote infomorphisms) has an underlying *diagram* (the shaded part of Figure 30), an *opvertex* classification ***A***, and a pair of infomorphisms called *opfirst* and *opsecond*, whose common target classification is the opvertex and whose source classifications are the target classifications of the infomorphisms in the span. The opfirst and opsecond infomorphisms form a commutative diagram with the span. A pushout cocone is the very special case of a colimiting cocone under a span. The term '`cocone`' denotes the *pushout cocone* collection. The term '`cocone-diagram`' represents the underlying diagram.

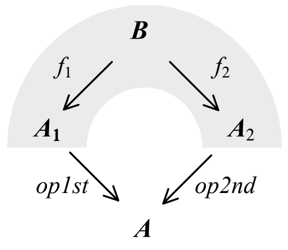


**Figure 30: Pushout Cocone**

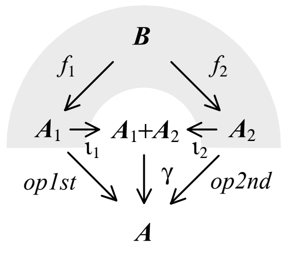
```
(15) (KIF$collection cocone)

(16) (KIF$function cocone-diagram)
     (= (KIF$source cocone-diagram) cocone)
     (= (KIF$target cocone-diagram) diagram)

(17) (KIF$function opvertex)
     (= (KIF$source opvertex) cocone)
     (= (KIF$target opvertex) CLS$classification)

(19) (KIF$function opfirst)
     (= (KIF$source opfirst) cocone)
     (= (KIF$target opfirst) CLS.INFO$infomorphism)
     (forall (?s (cocone ?s))
        (and (= (CLS.INFO$source (opfirst ?s)) (classification1 (cocone-diagram ?s)))
             (= (CLS.INFO$target (opfirst ?s)) (opvertex ?s))))
```

```
(20) (KIF$function opsecond)
     (= (KIF$source opsecond) cocone)
     (= (KIF$target opsecond) CLS.INFO$infomorphism)
     (forall (?s (cocone ?s))
        (and (= (CLS.INFO$source (opsecond ?s)) (classification2 (cocone-diagram ?s)))
             (= (CLS.INFO$target (opsecond ?s)) (opvertex ?s))))

(21) (forall (?s (cocone ?s))
        (= (CLS.INFO$composition [(first (cocone-diagram ?s)) (opfirst ?s)])
           (CLS.INFO$composition [(second (cocone-diagram ?s)) (opsecond ?s)])))
```

- The *binary-coproduct cocone* underlying any cocone (pushout diagram) is named.

```
(22) (KIF$function binary-coproduct-cocone)
     (= (KIF$source binary-coproduct-cocone) cocone)
     (= (KIF$target binary-coproduct-cocone) CLS.COL.COPRD$cocone)
     (forall (?s (cocone ?s))
        (and (= (CLS.COL.COPRD$cocone-diagram (binary-coproduct-cocone ?r))
                (pair (cocone-diagram ?s)))
             (= (CLS.COL.COPRD$opvertex (binary-coproduct-cocone ?r)) (opvertex ?s))
             (= (CLS.COL.COPRD$opfirst (binary-coproduct-cocone ?r)) (opfirst ?s))
             (= (CLS.COL.COPRD$opsecond (binary-coproduct-cocone ?r)) (opsecond ?s))))
```

- The *coequalizer cocone* function maps a pushout cocone of infomorphisms to the associated (`CLS.COL.COEQ`) coequalizer cocone of infomorphisms (see Figure 31, where arrows denote infomorphisms), which is the binary coproduct comediator of the opfirst and opsecond infomorphisms with respect to the coequalizer diagram of a cocone. This is the first step in the definition of the pushout comediator infomorphism. The following string of equalities demonstrates that this cocone is well-defined.



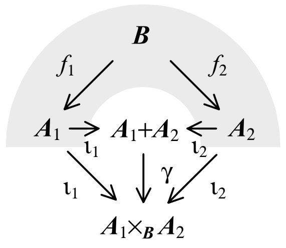
$$\begin{tikzcd} & \boldsymbol{B} \arrow[dl, "f_1"'] \arrow[dr, "f_2"] & \\ \boldsymbol{A}_1 \arrow[r, "\iota_1"'] \arrow[dr, "op1st"'] & \boldsymbol{A}_1+\boldsymbol{A}_2 \arrow[d, "\gamma"] & \boldsymbol{A}_2 \arrow[l, "\iota_2"] \arrow[dl, "op2nd"] \\ & \boldsymbol{A} & \end{tikzcd}$$

**Figure 31: Coequalizer Cocone**

$$f_1 \cdot \iota_1 \cdot \gamma = f_1 \cdot op1st = f_2 \cdot op1st = f_2 \cdot \iota_2 \cdot \gamma$$

```
(23) (KIF$function coequalizer-cocone)
     (= (KIF$source coequalizer-cocone) cocone)
     (= (KIF$target coequalizer-cocone) CLS.COL.COEQ$cocone)
     (forall (?s (cocone ?s))
         (and (= (CLS.COL.COEQ$cocone-diagram (coequalizer-cocone ?s))
                 (coequalizer-diagram (cocone-diagram ?s)))
              (= (CLS.COL.COEQ$opvertex (coequalizer-cocone ?s))
                 (opvertex ?s))
              (= (CLS.COL.COEQ$infomorphism (coequalizer-cocone ?s))
                 (CLS.COL.COPRD$comediator (binary-coproduct-cocone ?r)))))
```

- The KIF function '`colimiting-cocone`' maps a span to its pushout (colimiting pushout cocone) (see Figure 32, where arrows denote infomorphisms). For convenience of reference, we define three terms that represent the components of this pushout cocone. The opvertex of the pushout cocone is a specific *pushout* classification, which comes equipped with two *projection* infomorphisms. The last axiom expresses concreteness of the colimit – it expresses pushouts in terms of coproducts and coequalizers: the colimit of the coequalizer diagram is (not just isomorphic but) equal to the pushout; likewise, the compositions of the coproduct injections of the pair diagram with the canon of the coequalizer diagram are equal to the pushout injections.



$$\begin{tikzcd} & \boldsymbol{B} \arrow[dl, "f_1"'] \arrow[dr, "f_2"] & \\ \boldsymbol{A}_1 \arrow[r, "\iota_1"'] \arrow[dr, "\iota_1"'] & \boldsymbol{A}_1+\boldsymbol{A}_2 \arrow[d, "\gamma"] & \boldsymbol{A}_2 \arrow[l, "\iota_2"] \arrow[dl, "\iota_2"] \\ & \boldsymbol{A}_1 \times_{\boldsymbol{B}} \boldsymbol{A}_2 & \end{tikzcd}$$

**Figure 32: Colimiting Cocone**

```
(24) (KIF$function colimiting-cocone)
     (= (KIF$source colimiting-cocone) diagram)
     (= (KIF$target colimiting-cocone) cocone)
     (forall (?r (diagram ?r))
         (= (cocone-diagram (colimiting-cocone ?r)) ?r))

(25) (KIF$function colimit)
     (KIF$function pushout)
```

```
          (= pushout colimit)
          (= (KIF$source colimit) diagram)
          (= (KIF$target colimit) CLS$classification)

     (26) (KIF$function injection1)
          (= (KIF$source injection1) diagram)
          (= (KIF$target injection1) CLS.INFO$infomorphism)

     (27) (KIF$function injection2)
          (= (KIF$source injection2) diagram)
          (= (KIF$target injection2) CLS.INFO$infomorphism)

     (28) (forall (?r (diagram ?r))
              (and (= (colimit ?r) (opvertex (colimiting-cocone ?r)))
                   (= (CLS.INFO$source (injection1 ?r)) (classification1 ?r))
                   (= (CLS.INFO$target (injection1 ?r)) (colimit ?r))
                   (= (injection1 ?r) (opfirst (colimiting-cocone ?r)))
                   (= (CLS.INFO$source (injection2 ?r)) (classification2 ?r))
                   (= (CLS.INFO$target (injection2 ?r)) (colimit ?r))
                   (= (injection2 ?r) (opsecond (colimiting-cocone ?r)))))

     (29) (forall (?r (diagram ?r))
              (and (= (colimit ?r)
                      (CLS.COL.COEQ$coequalizer (coequalizer-diagram ?r)))
                   (= (injection1 ?r)
                      (CLS.INFO$composition
                          [(CLS.COL.COPRD$injection1 (pair ?r))
                           (CLS.COL.COEQ$canon (coequalizer-diagram ?r))]))
                   (= (injection2 ?r)
                      (CLS.INFO$composition
                          [(CLS.COL.COPRD$injection2 (pair ?r))
                           (CLS.COL.COEQ$canon (coequalizer-diagram ?r))]))))
```

o The following two axioms are the necessary conditions that the instance and type quasi-functors preserve concrete colimits. These ensure that both this pushout and its two pushout injection infomorphisms are specific.

```
     (30) (forall (?r (diagram ?r))
              (and (= (instance-cone (colimiting-cocone ?r))
                      (SET.LIM.PBK$limiting-cone (instance-diagram ?r)))
                   (= (CLS$instance (colimit ?r))
                      (SET.LIM.PBK$limit (instance-diagram ?r)))
                   (= (CLS.INFO$instance (injection1 ?r))
                      (SET.LIM.PBK$projection1 (instance-diagram ?r)))
                   (= (CLS.INFO$instance (injection2 ?r))
                      (SET.LIM.EQU$projection2 (instance-diagram ?r)))))

     (31) (forall (?r (diagram ?r))
              (and (= (type-cocone (colimiting-cocone ?r))
                      (SET.COL.PSH$colimiting-cocone (type-diagram ?r)))
                   (= (CLS$type (colimit ?r))
                      (SET.COL.PSH$colimit (type-diagram ?r)))
                   (= (CLS.INFO$type (injection1 ?r))
                      (SET.COL.PSH$injection1 (instance-diagram ?r)))
                   (= (CLS.INFO$type (injection2 ?r))
                      (SET.COL.PSH$injection2 (instance-diagram ?r)))))
```

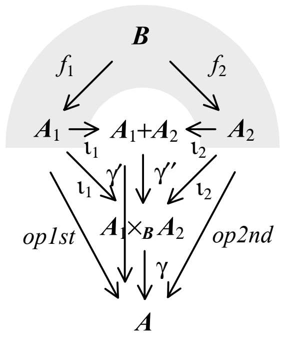

$$\begin{tikzcd}
 & & \boldsymbol{B} \arrow[dl, "f_1"'] \arrow[dr, "f_2"] & & \\
 \boldsymbol{A}_1 \arrow[r, "\iota_1"'] \arrow[drr, "\iota_1"] \arrow[ddrr, "op1st"'] & \boldsymbol{A}_1{+}\boldsymbol{A}_2 \arrow[dr, "\gamma'"'] \arrow[ddr, "\gamma'"] & & \boldsymbol{A}_1{+}\boldsymbol{A}_2 & \boldsymbol{A}_2 \arrow[l, "\iota_2"] \arrow[dll, "\iota_2"'] \arrow[ddll, "op2nd"] \\
 & & \boldsymbol{A}_1 \times_{\boldsymbol{B}} \boldsymbol{A}_2 \arrow[d, "\gamma"] & & \\
 & & \boldsymbol{A} & &
\end{tikzcd}$$

**Figure 33: Comediator**

o The *comediator* infomorphism, from the pushout of a span to the opvertex of a cocone over the span (see Figure 33, where arrows denote infomorphisms), is the unique infomorphism that commutes with opfirst and opsecond. This is defined abstractly by using a definite description, and is defined concretely as the comediator of coequalizer cocone.

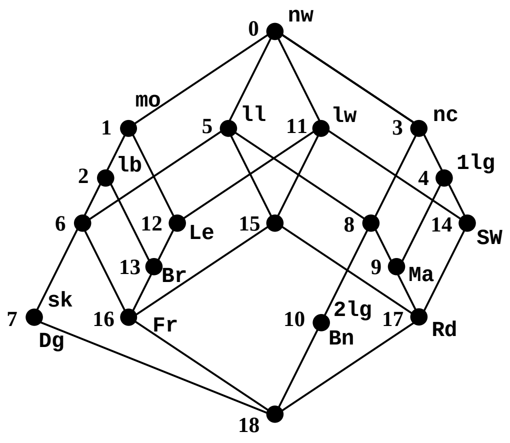
```
     (32) (KIF$function comediator)
          (= (KIF$source comediator) cocone)
          (= (KIF$target comediator) CLS.INFO$infomorphism)
          (forall (?s (cocone ?s))
              (and (= (CLS.INFO$source (comediator ?s)) (colimit (cocone-diagram ?s)))
                   (= (CLS.INFO$target (comediator ?s)) (opvertex ?s))
                   (= (comediator ?s)
                      (the (?m (CLS.INFO$infomorphism ?m))
                          (and (= (composition [(injection1 (cocone-diagram ?s)) ?m])
                                  (opfirst ?s))
                               (= (composition [(injection2 (cocone-diagram ?s)) ?m])
                                  (opsecond ?s)))))))

     (33) (forall (?s (cocone ?s))
              (= (comediator ?s)
                 (CLS.COL.COEQ$comediator (coequalizer-cocone ?s))))
```

## *Examples*

### The Living Classification

- The *Living Classification* is a tiny dataset, which exists within a conceptual universe of living organisms. This classification listed below consists of eight organisms (plants and animals), and nine of their properties. The organisms are the *instances* of the classification, and the properties are the *types*. The classification relation is presented as a Boolean matrix in the lower-right table. The *Living Concept Lattice*, which contains 19 formal concepts, is visualized in the upper-left image.

#### Concept Lattice



#### Type Class

| | | |
|---|---|---|
| 0 | nw | needs water |
| 1 | lw | lives in water |
| 2 | ll | lives on land |
| 3 | nc | needs chlorophyll |
| 4 | 2lg | 2 leaf germination |
| 5 | 1lg | 1 leaf germination |
| 6 | mo | is motile |
| 7 | lb | has limbs |
| 8 | sk | suckles young |

#### Instance Class

| | | |
|---|---|---|
| 0 | Le | Leech |
| 1 | Br | Bream |
| 2 | Fr | Frog |
| 3 | Dg | Dog |
| 4 | SW | Spike Weed |
| 5 | Rd | Reed |
| 6 | Bn | Bean |
| 7 | Ma | Maize |

#### Classification Relation

| | nw | lw | ll | nc | 2lg | 1lg | mo | lb | sk |
|---|---|---|---|---|---|---|---|---|---|
| Le | × | × | | | | | × | | |
| Br | × | × | | | | | × | × | |
| Fr | × | × | × | | | | × | × | |
| Dg | × | | × | | | | × | × | × |
| SW | × | × | | × | | × | | | |
| Rd | × | × | × | × | | × | | | |
| Bn | × | | × | × | × | | | | |
| Ma | × | | × | × | | × | | | |

The following table lists the formal concepts of the Living concept lattice in terms of their extent, intent, instance generators and type generators.

| Formal Concepts | | | | |
|---|---|---|---|---|
| | Generators | | | |
| Index | Objects | Attributes | Extent | Intent |
| 0 | | needs water | {Leech, Bream, Frog, Dog, Spike | {needs water } |

| | | | | |
|---|---|---|---|---|
| | | | Weed, Reed, Bean, Maize} | |
| 1 | | is motile | {Leech, Bream, Frog, Dog} | {needs water, is motile } |
| 2 | | has limbs | {Bream, Frog, Dog} | {needs water, is motile, has limbs } |
| 3 | | needs chlorophyll | {Spike Weed, Reed, Bean, Maize} | {needs water, needs chlorophyll } |
| 4 | | 1 leaf germination | {Spike Weed, Reed, Maize} | {needs water, needs chlorophyll, 1 leaf germination } |
| 5 | | lives on land | {Frog, Dog, Reed, Bean, Maize} | {needs water, lives on land } |
| 6 | | | {Frog, Dog } | {needs water, lives on land, is motile, has limbs } |
| 7 | Dog | suckles young | {Dog } | {needs water, lives on land, is motile, has limbs, suckles young } |
| 8 | | | {Reed, Bean, Maize } | {needs water, lives on land, needs chlorophyll } |
| 9 | Maize | | {Reed, Maize } | {needs water, lives on land, needs chlorophyll, 1 leaf germination } |
| 10 | Bean | 2 leaf germination | {Bean } | {needs water, lives on land, needs chlorophyll, 2 leaf germination } |
| 11 | | lives in water | {Leech, Bream, Frog, Spike Weed, Reed } | {needs water, lives in water } |
| 12 | Leech | | {Leech, Bream, Frog } | {needs water, lives in water, is motile } |
| 13 | Bream | | {Bream, Frog } | {needs water, lives in water, is motile, has limbs } |
| 14 | Spike Weed | | {Spike Weed, Reed } | {needs water, lives in water, needs chlorophyll, 1 leaf germination } |
| 15 | | | {Frog, Reed } | {needs water, lives in water, lives on land } |
| 16 | Frog | | {Frog} | {needs water, lives in water, lives on land, is motile, has limbs } |
| 17 | Reed | | {Reed } | {needs water, lives in water, lives on land, needs chlorophyll, 1 leaf germination } |
| 18 | | | {} = ∅ | {needs water, lives in water, lives on land, needs chlorophyll, 2 leaf germination, 1 leaf germination, is motile, has limbs, suckles young } |

○ The following KIF represents the *Living Classification*.

```
(CLS$Classification Living)
((CLS$type Living) needs-water)
((CLS$type Living) lives-in-water)
((CLS$type Living) lives-on-land)
((CLS$type Living) needs-chlorophyll)
((CLS$type Living) 2-leaf-germination)
((CLS$type Living) 1-leaf-germination)
((CLS$type Living) is-motile)
((CLS$type Living) has-limbs)
```

```
((CLS$type Living) suckles-young)
((CLS$instance Living) Leech)
((CLS$instance Living) Bream)
((CLS$instance Living) Frog)
((CLS$instance Living) Dog)
((CLS$instance Living) Spike-Weed)
((CLS$instance Living) Reed)
((CLS$instance Living) Bean)
((CLS$instance Living) Maize)
...
(Living Leech needs-water)
(Living Leech lives-in-water)
(not (Living Leech lives-on-land))
(not (Living Leech needs-chlorophyll))
(not (Living Leech 2-leaf-germination))
(not (Living Leech 1-leaf-germination))
(Living Leech is-motile)
(not (Living Leech has-limbs))
(not (Living Leech suckles-young))
...
```

○ The *Living Lattice* is the concept lattice of the Living Classification.

```
(CL$concept-lattice Living-Lattice)
(= Living-Lattice (CLS.CL$concept-lattice Living))
```

## The Dictionary Classification

Here are examples of classifications and infomorphisms taken from the text *Information Flow: The Logic of Distributed Systems* by Barwise and Seligman.

○ The following KIF represents the *Webster Classification* on page 70 of Barwise and Seligman. This classification, which is (a small part of) the classification of English words according to parts of speech as given in Webster's dictionary, is diagrammed on the right.

**Table 4: Webster Classification**

| **Webster** | Noun | Int-Vb | Tr-Vb | Adj |
|---|---|---|---|---|
| bet | 1 | 1 | 1 | 0 |
| eat | 0 | 1 | 1 | 0 |
| fit | 1 | 1 | 1 | 1 |
| friend | 1 | 0 | 1 | 0 |
| square | 1 | 0 | 1 | 1 |
| … | | … | | |

```
(CLS$classification Webster)
((CLS$type Webster) Noun)
((CLS$type Webster) Intransitive-Verb)
((CLS$type Webster) Transitive-Verb)
((CLS$type Webster) Adjective)
((CLS$instance Webster) bet)
((CLS$instance Webster) eat)
((CLS$instance Webster) fit)
((CLS$instance Webster) friend)
((CLS$instance Webster) square)
...
(Webster bet Noun)
(Webster bet Intransitive-Verb)
(Webster bet Transitive-Verb)
(not (Webster bet Adjective))
(not (Webster eat Noun))
(Webster eat Intransitive-Verb)
(Webster eat Transitive-Verb)
(not (Webster fit Adjective))
(Webster fit Noun)
(Webster fit Intransitive-Verb)
(Webster fit Transitive-Verb)
(Webster fit Adjective)
(Webster friend Noun)
(not (Webster friend Intransitive-Verb))
(Webster friend Transitive-Verb)
(not (Webster friend Adjective))
(Webster square Noun)
(not (Webster square Intransitive-Verb))
(Webster square Transitive-Verb)
(Webster square Adjective)
```

```
(CLS$classification Webster)
((CLS$type Webster) Noun)
((CLS$type Webster) Intransitive-Verb)
((CLS$type Webster) Transitive-Verb)
((CLS$type Webster) Adjective)
((CLS$instance Webster) bet)
((CLS$instance Webster) eat)
((CLS$instance Webster) fit)
((CLS$instance Webster) friend)
((CLS$instance Webster) square)
...
(Webster bet Noun)
(Webster bet Intransitive-Verb)
(Webster bet Transitive-Verb)
(not (Webster bet Adjective))
(not (Webster eat Noun))
(Webster eat Intransitive-Verb)
(Webster eat Transitive-Verb)
(not (Webster fit Adjective))
(Webster fit Noun)
(Webster fit Intransitive-Verb)
(Webster fit Transitive-Verb)
(Webster fit Adjective)
(Webster friend Noun)
(not (Webster friend Intransitive-Verb))
(Webster friend Transitive-Verb)
(not (Webster friend Adjective))
(Webster square Noun)
(not (Webster square Intransitive-Verb))
(Webster square Transitive-Verb)
(Webster square Adjective)
...
```

○ The following KIF represents the infomorphism defined on page 73 of Barwise and Seligman. This represents the way that punctuation at the end of a sentence carries information about the type of the sentence. The infomorphism is from a *Punctuation* classification to a *Sentence* classification. The instances of *Punctuation* are the inscriptions of the punctuation marks of English. These marks are classified by the terms '`Period`, '`Exclamation-Mark`', '`Question-Mark`', '`Comma`', etc. The instances of *Sentence* are inscriptions of grammatical sentences of English. There are but three types of *Sentence*: '`Declarative`', '`Question`', and '`Other`'. The instance function of the infomorphism assigns to each sentence its own terminating punctuation mark. The type function of the infomorphism assigns '`Declarative`' to '`Period`' and '`Exclamation-Mark`', '`Question`' to '`Question-Mark`', and '`Other`' to other types of *Punctuation*. The fundamental property of this infomorphism is the requirement that a sentence be of the type indicated by its punctuation.

Let '`yakity-yak`' denote the command "Take out the papers and the trash!" with its punctuation symbol '`yy-punc`' being the exclamation symbol at the end of the sentence. Let '`gettysburg1`' denote the statement that "Fourscore and seven years ago our fathers brought forth on this continent a new nation, conceived in liberty and dedicated to the proposition that all men are created equal." with its punctuation symbol '`g1-punc`' being the period at the end of the sentence. Let '`angels`' denote the question "How many angels can fit on the head of a pin?" with its punctuation symbol '`ag-punc`' being the question mark at the end of the sentence.

```
(CLS$classification Punctuation)
((CLS$type Punctuation) Period)
((CLS$type Punctuation) Exclamation-Mark)
((CLS$type Punctuation) Question-Mark)
((CLS$type Punctuation) Comma)
((CLS$instance Punctuation) yy-punc)
((CLS$instance Punctuation) g1-punc)
((CLS$instance Punctuation) ag-punc)
...
(not (Punctuation yy-punc Period))
(Punctuation yy-punc Exclamation-Mark)
(not (Punctuation yy-punc Question-Mark))
...
```

```
(CLS$classification Sentence)
((CLS$type Sentence) Declarative)
((CLS$type Sentence) Question)
((CLS$type Sentence) Other)
((CLS$instance Sentence) yakity-yak)
((CLS$instance Sentence) gettysburg1)
((CLS$instance Sentence) angels)
...
(Sentence gettysburg1 Declarative)
(not (Sentence gettysburg1 Question))
(not (Sentence gettysburg1 Other))
...
(CLS.INFO$infomorphism punct-type)
(= (CLS.INFO$source punct-type) Punctuation)
(= (CLS.INFO$target punct-type) Sentence)

(= ((CLS.INFO$instance punct-type) yakity-yak) yy-punc)
(= ((CLS.INFO$instance punct-type) gettysburg1) g1-punc)
...
(= ((CLS.INFO$type punct-type) Period) Declarative)
(= ((CLS.INFO$type punct-type) Exclamation-Mark) Declarative)
(= ((CLS.INFO$type punct-type) Question-Mark) Question)
(= ((CLS.INFO$type punct-type) Comma) Other)
...
```

## The Truth Classification

o The *truth classification* of a first-order language ***L*** is the large classification, whose instances are ***L***-structures, whose types are ***L***-sentences, and whose classification relation is satisfaction. Here we represent the truth classification in an external namespace. Note that the source is a class, whereas the target is a collection – rather unusual. The image should then be just a class.

```
(KIF$function truth-classification)
(= (KIF$source truth-classification) lang$language)
(= (KIF$target truth-classification) CLS$classification)
(forall (?l (lang$language ?l))
    (and (= (CLS$instance (truth-classification ?l))  (MOD$fiber ?l))
         (= (CLS$type (truth-classification ?l))      (lang$sentence ?l))
         (= (truth-classification ?l) (MOD$satisfaction ?l))))
```

○ The *truth concept lattice* is the concept lattice of the truth classification. This large complete lattice can function as the appropriate "lattice of ontological theories" for a modular SUO architecture. A formal concept in this lattice has an intent that is a closed theory (set of sentences) and an extent that is the class of all models for that theory. The intent (theory) of the join of two formal concepts is the intersection of the intents (theories) of the formal concepts. The intent (theory) of the meet of two formal concepts is the type-closure of the union of the intents (theories) of the formal concepts, or the theory of the common models.

```
(KIF$function truth-concept-lattice)
(= (KIF$source truth-concept-lattice) lang$language)
(= (KIF$target truth-concept-lattice) CL$concept-lattice)
(forall (?l (lang$language ?l))
    (= (truth-concept-lattice ?l)
       (CLS.CL$concept-lattice (truth-classification ?l))))
```