The IFF Type Namespace

Robert E. Kent

December 18, 2007

Contents

1 The Type Kernel **2**
1.1 Introduction 2
1.2 Type Sets 9
1.3 Type Functions 21
1.3.1 Function Morphisms 35
1.4 Type Spans 40
1.4.1 Type Endospans 55
1.4.2 Span Morphisms 57
1.5 Type Predicates 65
1.6 Type Relations 73
1.6.1 Type Endorelations 85

<table>
<tr><td colspan="2">ordinary element pairs
$(x_0, x_1) \in X_0 \times X_1$</td><td>generalized element pairing
$Y \xrightarrow{\langle x_0, x_1 \rangle} X_0 \times X_1$</td></tr>
<tr><td>relations as inclusions
$\mathsf{ext}(r) \subseteq X_0 \times X_1$</td><td colspan="2">relations as injections
$\mathsf{ext}(r) \overset{r}{\hookrightarrow} X_0 \times X_1$</td></tr>
<tr><td colspan="3">relation membership in terms of</td></tr>
<tr><td>set membership</td><td>function application</td><td>function composition</td></tr>
<tr><td colspan="3">$r(x_0, x_1)$ when</td></tr>
<tr><td>$(x_0, x_1) \in \mathsf{ext}(r)$</td><td>$\exists_{y \in \mathsf{ext}(r)}\, r(y) = (x_0, x_1)$</td><td>$\exists_{y : Y \to \mathsf{ext}(r)}\, y \cdot r = \langle x_0, x_1 \rangle$</td></tr>
</table>

Table 9: Relations, Elements and Membership

1.6 Type Relations

Introduction. Just as sets represent the nouns in natural language expressions, and predicates represent the adjectives, so also relations represent the verbs. In the expression "Elizabeth marries Darcy", the verb "marry" links the subject "Elizabeth" to the direct object "Darcy". The semantics of this expression consists of a pair of individuals asserted to be a member of a binary relation. In mathematics and knowledge engineering, relations, element pairs and relation membership have several representations, as indicated in Table 9. Relations can be represented as subsets (inclusions) or injections, element pairs can be represented as ordinary (global) element pairs or generalized element (function) pairing, and the representation of relation membership varies accordingly[12].

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
(r x0 x1)
```

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

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

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

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

This equivalence can be expressed in terms of predicates

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

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

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

[a] equivalently, "$r \in \mathsf{rel}, \sigma_0(r) = X_0, \sigma_1(r) = X_1, \varepsilon(r) = Y$"

Table 10: IFF Relational Notation

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

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

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

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

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

$$\begin{array}{ccc} \mathsf{ext}(r) & \xrightarrow{r} & X_0{\times}X_1 \\ \downarrow & & \downarrow \\ \mathsf{ext}(s) & \xrightarrow[s]{} & Y_0{\times}Y_1 \end{array}$$

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

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

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

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

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

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

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

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

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

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

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

Composition is associative.

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

Composition is surjective (see identity properties below).

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

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

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

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

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

The identity satisfies two unit laws with respect to composition.

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

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

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

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

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

Conversion.

$$\begin{array}{ccc} \mathsf{rel} & \xrightarrow{(\dot{-})} & \mathsf{pred} \\ {\scriptstyle \sigma_{01}}\downarrow & & \downarrow{\scriptstyle \gamma} \\ \mathsf{set}^2 & \xrightarrow[\times]{} & \mathsf{set} \end{array}$$

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

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

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

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

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

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

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

Every type relation $r : X_0 \rightharpoondown X_1$ has an associated *span* $\hat{r} = (\pi^r_0 : X_0 \leftarrow \mathsf{ext}(r) \rightarrow X_1 : \pi^r_1)$, whose pairing is the injection $r = (\pi^r_0, \pi^r_1) : \mathsf{ext}(r) \rightarrow X_0{\times}X_1$.

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

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

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

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

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

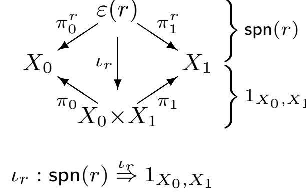


Figure 12: Injection of a Relation

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

Any type relation $r : X_0 \rightharpoondown X_1$ with injective function $\mathsf{ext}(r) \overset{r}{\hookrightarrow} X_0{\times}X_1$ has an *opposite* type relation $r^{\propto} : X_1 \rightharpoondown X_0$ with injective function $\mathsf{ext}(r) \overset{r}{\hookrightarrow} X_0{\times}X_1 \overset{\tau}{\to} X_1{\times}X_0$, where τ is the binary product twist bijection.

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

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

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

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

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

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

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

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

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

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

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

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

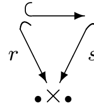
```
(forall ((relation ?r) ((set0 ?r) ?x0) ((set1 ?r) ?x1))
    (<=> (((fiber01 ?r) ?x0) ?x1)
         (?r ?x0 ?x1)))

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

The two fibers are equivalent under involution.

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

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



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

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

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

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

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

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

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

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

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

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

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

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

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

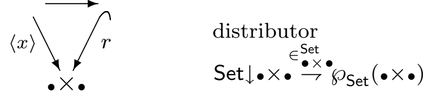
Any relation is equivalent to its canonically strict associate, $r \equiv [r]$. Hence, for any type relation r, the equivalence class $[r]$ has the canon as canonical representative.

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

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

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

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



$\mathsf{Set}{\downarrow}\bullet{\times}\bullet \overset{\in^{\mathsf{Set}}_{\bullet\times\bullet}}{\rightharpoondown} \wp_{\mathsf{Set}}(\bullet{\times}\bullet)$

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

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

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

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

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

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

$$r \subseteq s \;\; \textit{iff} \;\; \forall_{x \in (X_0,X_1)}(x \in r \Rightarrow x \in s) \qquad \subseteq = \in \backslash \in$$

Hence, the usual relationship holds between inclusion and membership.

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

We name the unique *proof* function for membership.

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

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

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

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

1.6.1 Type Endorelations

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

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

There is a common component type *set*.

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

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

- A emphreflexive type endorelation is one that contains the identity endorelation $1_X \subseteq r$. Since there is no notion of predicate specified at the IFF level, at the type level we use the differentia set of the reflexive relation[13] to indirectly specify it.
- A emphtransitive type endorelation is one that contains the relational composition of it with itself $r \circ r \subseteq r$.
- A emphsymmetric type endorelation is one whose transpose is contained in it $r^{\propto} \subseteq r$.
- An *antisymmetric* type endorelation is one where the meet of it with its transpose is a subrelation of the identity $r \wedge_X r^{\propto} \subseteq 1_X$.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Our two standard references for the IFF are the books: *Sets for Mathematics* (2003) by F. William Lawvere and Robert Rosebrugh [1] and *Categories for the Working Mathematician* (1971) by Saunders Mac Lane [2].

References

[1] F. William Lawvere and Robert Rosebrugh. *Sets for Mathematics.* Cambridge University Press, 2003.

[2] Saunders Mac Lane. *Categories for the Working Mathematician.* Springer-Verlag, 1971.