The Namespace of Models

Table 1 lists the terminology in the model namespace of the IFF Model Theory Ontology.

Table 1: Terminology for the Model Namespace

	Class	Function	Other
mod	model = structure	reference signature reference-arity arity entity relation variable universe tuple instance type	
		set-pair function-pair induction compatible strict-compatible trim	
		primitive-extent negation-extent connective-operation connective-extent quantifier-operation quantifier-extent strict-substitution substitution substitution-extent extent-tuple extent expression-classification expression embedding	
		relationally-represents relationally-satisfies strongly-relationally-satisfies expressively-represents expressively-satisfies strongly-expressively-satisfies satisfies	
		theorem theory true maximal-theory logic	
		language-intent-morphism language-intent-infomorphism language-intent **[counit of** $\omega = \kappa \cdot \mu$**]**	
		formal-tuple formal-relation theory-intent-function theory-intent-infomorphism theory-intent **[counit of** $\mu = \lambda \cdot \pi$**]**	
		empty full	
		case indication projection comediator spanmodel	
mod **.mor**	infomorphism = structure-morphism simple	source target reference signature reference-arity arity entity relation variable universe tuple composition identity instance type	composable-opspan composable
		expression-infomorphism expression	
		theorem-function theory-function maximal-theory logic	
		empty full	

Models

`mod`

A "possible world" or *model* for a type language provides an interpretation for all types. It traditionally associates sets (domains) with entity types and relations with relation types. These associations must respect the various typings. The notion of model introduced in the IFF Model Theory Ontology is framed in terms of the Information Flow notion of a classification along with a suitable notion of a hypergraph. This is a new formulation of model, and introduced here for the first time. However, these IFF models can be placed in the traditional perspective. This section axiomatizes the namespace for models and their morphisms. These form a category called Model.
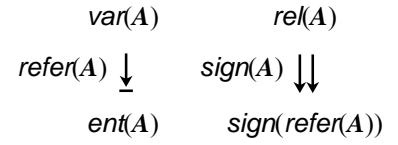
$$var(A) \qquad rel(A)$$
$$refer(A) \downarrow \qquad sign(A) \downdownarrows$$
$$ent(A) \qquad sign(refer(A))$$

Figure 1: Model

○   A *model* (model-theoretic *structure*) $A = \langle refer(A), sign(A) \rangle$ is a hypergraph of classifications – a two dimensional construction (Figure 1) consisting of

–   a *reference* semidesignation $\tau_A = refer(A) = \langle inst(refer(A)), typ(refer(A)) \rangle : var(A) \rightarrowtail ent(A)$ and

–   a *signature* designation $\partial_A = sign(A) = \langle inst(sign(A)), typ(sign(A)) \rangle : rel(A) \Rightarrow sign(refer(A))$,

where the signature classification of the reference semidesignation is the target classification of the signature designation
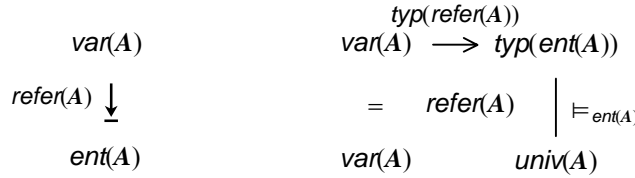
$$sign(refer(A)) = tgt(sign(A)).$$

$$var(A)$$
$$refer(A) \downarrow$$
$$ent(A)$$

$$\begin{array}{ccc} & typ(refer(A)) & \\ var(A) & \longrightarrow & typ(ent(A)) \\ = & refer(A) & \Big| \vDash_{ent(A)} \\ var(A) & & univ(A) \end{array}$$

Figure 2a: Reference Semidesignation – abstract

Figure 2b: Reference Semidesignation – details

$$rel(A)$$
$$sign(A) \downdownarrows$$
$$sign(refer(A))$$

$$\begin{array}{ccc} & typ(sign(A)) & \\ typ(rel(A)) & \longrightarrow & sign(typ(refer(A))) \\ \vDash_{rel(A)} \Big| & sign(A) & \Big| \vDash_{sign(refer(A))} \\ tuple(A) & \longrightarrow & tuple(refer(A)) \\ & inst(sign(A)) & \end{array}$$
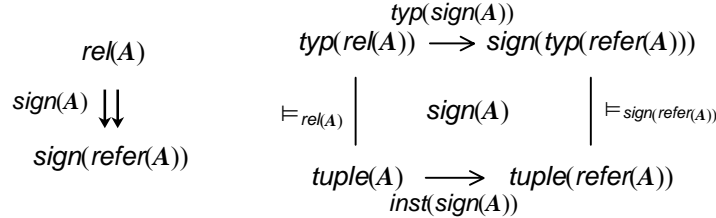
Figure 3a: Signature Designation - abstract

Figure 3b: Signature Designation - details

```
(1) (SET$class model)
    (SET$class structure)
    (= structure model)

(2) (SET.FTN$function reference)
    (= (SET.FTN$source reference) model)
    (= (SET.FTN$target reference) cls.sdsgn$semidesignation)

(3) (SET.FTN$function signature)
    (= (SET.FTN$source signature) model)
    (= (SET.FTN$target signature) cls.dsgn$designation)

(4) (= (SET.FTN$composition [reference cls.sdsgn$signature])
       (SET.FTN$composition [signature cls.dsgn$target]))
```
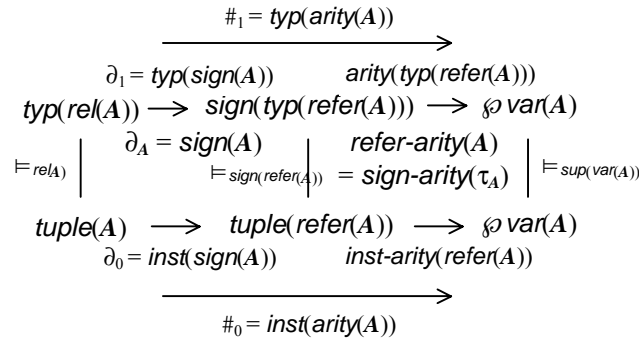
○

$$\#_1 = typ(arity(A))$$

$$\partial_1 = typ(sign(A)) \qquad arity(typ(refer(A)))$$

$$typ(rel(A)) \longrightarrow sign(typ(refer(A))) \longrightarrow \wp\, var(A)$$

$$\partial_A = sign(A) \qquad refer\text{-}arity(A)$$

$$\vDash_{rel(A)} \Big| \qquad \vDash_{sign(refer(A))} = sign\text{-}arity(\tau_A) \qquad \Big| \vDash_{sup(var(A))}$$

$$tuple(A) \longrightarrow tuple(refer(A)) \longrightarrow \wp\, var(A)$$

$$\partial_0 = inst(sign(A)) \qquad inst\text{-}arity(refer(A))$$

$$\#_0 = inst(arity(A))$$

Diagram 1: Signature, reference-arity and arity designations

○ For convenience of theoretical presentation, we introduce model terminology for the composition of the reference function and the arity function for semidesignations: a *reference-arity* function *refer-arity*$(A) = sign\text{-}arity(refer(A))$.

```
(5) (SET.FTN$function reference-arity)
    (= (SET.FTN$source reference-arity) model)
    (= (SET.FTN$target reference-arity) cls.dsgn$designation)
    (= reference-arity (SET.FTN$composition [reference cls.sdsgn$sign-arity]))
```

○ For any model *A* the composition (Diagram 1) of the signature designation with the reference arity designation defines an *arity* designation $\#_A = \langle \#_0, \#_1 \rangle = arity(A) = sign(A) \cdot refer\text{-}arity(A)$.

```
(6) (SET.FTN$function arity)
    (= (SET.FTN$source arity) model)
    (= (SET.FTN$target arity) cls.dsgn$designation)
    (forall (?a (model ?a))
        (= (arity ?a)
           (cls.dsgn$composition [(signature ?a) (reference-arity ?a)])))
```

○ For convenience of practical reference, we introduce additional model terminology for the source and target components of these designations. The source of the signature designation is called the *relation* classification of *A* and denoted *rel*(*A*). The classification of the reference semidesignation is called the *entity* classification of *A* and denoted *ent*(*A*). The set of the reference semidesignation is called the set of logical *variables* of *A* and denoted *var*(*A*). In summary, we provide terminology for the following sets/classifications:

– the *entity* classification $ent(A) = cla(refer(A))$,

– the *relation* classification $rel(A) = src(sign(A))$, and

– the *variable* set $var(A) = set(refer(A))$.

This results in the following presentations of the signature, arity and reference designations:

– *signature* designation $\partial_A = \langle \partial_0, \partial_1 \rangle = sign(A) : rel(A) \to sign(refer(A))$,

– *arity* designation $\#_A = \langle \#_0, \#_1 \rangle = arity(A) : rel(A) \to sup(var(A))$, and

– *reference* semidesignation $\tau_A = refer(A)$ with set $var(A)$ and classification $ent(A)$.

One canonical definition of the set of variables is the Cartesian product $var(A) = typ(ent(A)) \times Natno$. Then the reference type function is the projection function

$$typ(refer(A)) = \pi_1 : typ(ent(A)) \times Natno \to typ(ent(A)).$$

```
(7) (SET.FTN$function entity)
    (= (SET.FTN$source entity) model)
    (= (SET.FTN$target entity) cls$classification)
    (= entity (SET.FTN$composition [reference cls.sdsgn$classification]))
```

```
(8) (SET.FTN$function relation)
    (= (SET.FTN$source relation) model)
```

```
        (= (SET.FTN$target relation) cls$classification)
        (= relation (SET.FTN$composition [signature dsgn$source]))

(9) (SET.FTN$function variable)
    (= (SET.FTN$source variable) model)
    (= (SET.FTN$target variable) cls$classification)
    (= variable (SET.FTN$composition [reference cls.sdsgn$set]))
```

○   Let us define some further common terminology for models. The set $univ(A) = inst(ent(A))$ of entity instances is called the *universe* (or *universal domain*) for A. In the entity classification $ent(A)$, the elements of the universe are classified by entity types. Elements of the set of relational instances $tuple(A) = inst(rel(A))$ are called *tuples* or (*relational*) *arguments*. In the relation classification $rel(A)$, tuples are classified by relation types. Although we do not specify it, usually there is a subset of individuals $indiv(A) \subseteq univ(A)$, whose elements are called *individual designators*. We use some terminology from conceptual graphs here. Individual designators, thought of as identifiers for individuals, represent either objects or data values. A locator represents an object: a marker 'ISBN-0-521-58386-1', an indexical 'you' or a name 'K. Jon Barwise'. A data value is represented as a literal: a string 'xyz', a number '3.14159', a date '1776/07/04', etc.

○   Tuples in $r \in tuple(A)$ are abstract, and by themselves are typeless. They are associated with concrete reference tuples, and thus elements of the universe, via the tuple function

$$\partial_0 = inst(sign(A)) : tuple(A) \to tuple(refer(A)).$$

A concrete tuple $\partial_0(r) \in tuple(refer(A))$ is a tuple of entity instances $\partial_0(r) \in univ(A)^{arity(t)}$. Thus, it assigns elements of the universe (entity instances) to the variables in the subset $arity(r)$. The arity is associated with each abstract tuple $r \in tuple(A)$ via the arity function

$$\#_0 = inst(arity(A)) : tuple(A) \to \wp var(A).$$

○   The entity classification $ent(A) = \langle univ(A), typ(ent(A)), \vDash_{ent(A)} \rangle$ has the entity types of A as its types and the elements of the universe of A as its instances. The relation classification $rel(A) = \langle tuple(A), typ(rel(A)), \vDash_{rel(A)} \rangle$ has the relation types of A as its types and the tuples of A as its instances. As explained above, in the relation classification the abstract tuples map to concrete tuples. These are called *records* in the context of database relations: if tuple r has relational type ρ, $r \vDash_{rel(A)} \rho$, then the concrete tuple $\partial_0(r)$ has the signature type $\partial_1(\rho)$, $\partial_0(r) \vDash_{sign(ref(A))} \partial_1(\rho)$; this means that the arities are related as $arity(\rho) \subseteq arity(r)$, and $\partial_0(r)(j(x)) \vDash_{ent(A)} \partial_1(\rho)(x)$ for each variable $x \in arity(\rho)$. Of course, although a tuple r has a fixed arity $arity(r) = \{x_1, x_2, \dots x_m\}$ and a fixed signature $\partial_0(r) = \{\partial_0(r)(x) \mid x \in arity(r)\} = (\partial_0(r)(x_1), \partial_0(r)(x_2), \dots \partial_0(r)(x_m))$, a different relational classification incidence $r \vDash_{rel(A)} \rho'$ may assert $arity(\rho') \subseteq arity(r)$, and $\partial_0(r)(j'(x')) \vDash_{ent(A)} \partial_1(\rho')(x')$ for each variable $x' \in arity(\rho')$, where even the cardinalities (valences) differ $|arity(\rho)| \neq |arity(\rho')|$. For an example with the same valences but different arities, a tuple with the instance signature (Sam, 70) could represent an age or a height, corresponding to two different type signatures (Person, Years) and (Person, Inches).

○   Here is an example that partially illustrates how frames can be represented in the IFF Model Theory Ontology. Frames are used to represent objects, events, abstract relationships, etc. A frame has (at least) the following parts:

–   a *name*
–   zero or more *abstractions* or *superclasses* (classes to which the concept belongs)
–   zero or more *slots* or *attributes* that describe particular properties of the concept.

Frames use linguistic roles, such as 'agent', 'patient' and 'instrument' to fill verb-specific frame slots.

Table 2: Role Frame

frame: send	
role	*filler*
agent	Adam
patient	Eve
object	flowers
instrument	email

For example, the sentence "Adam sent the flowers to Eve by email." would fill the 'send' frame shown in Table 2. Here is one way to represent this in the IFF Model Theory Ontology:

- the frame name 'send' is a relation type,
- each role 'agent', 'patient', 'object' and 'instrument' is a variable (entity type name),
- the set of roles {'agent', 'patient', 'object', 'instrument'} is an arity,
- since this kind of frame is untyped, using a universal entity type 'Entity' the signature of the relation type 'send' is

(*agent*: Entity, *patient*: Entity, *object*: Entity, *instrument*: Entity),

- each role filler 'Adam', 'Eve', 'flowers' and 'email' is an element of the universe (entity instance),
- there is an abstract tuple of relation type 'send' which is assigned this tuple of instances,

(*agent*: Adam, *patient*: Eve, *object*: flowers, *instrument*: email),

- each role filling, such as '*agent*: Adam' is a pointwise classification incidence 'Adam' ⊨ '*agent*'.

So frame names are represented by relation type symbols, roles are represented by variables, and role fillers are represented by entity instances. The frame itself represents the classification incidence between the tuple and the relation type 'send'. Frame abstractions (sub-classification) can be specified by using sequents in the IF theories that correspond to classifications.

```
(10) (SET.FTN$function universe)
     (= (SET.FTN$source universe) model)
     (= (SET.FTN$target universe) set$set)
     (= universe (SET.FTN$composition [entity cls$instance]))

(11) (SET.FTN$function tuple)
     (= (SET.FTN$source tuple) model)
     (= (SET.FTN$target tuple) set$set)
     (= tuple (SET.FTN$composition [relation cls$instance]))
```
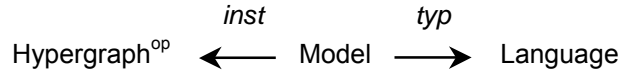
$$\text{Hypergraph}^{op} \xleftarrow{\quad inst \quad} \text{Model} \xrightarrow{\quad typ \quad} \text{Language}$$

Diagram 2: The Hypergraph Span of a Model

○   There is a derived view of a model as a "classification" of hypergraphs – the hypergraphs are classified componentwise. A model A with components

$$\langle refer(A), sign(A), arity(A), ent(A), rel(A), var(A), univ(A), tuple(A) \rangle$$

defines two hypergraphs (Diagram 2), an *instance* hypergraph $inst(A)$ and a *type* language $typ(A)$. The components of the hypergraphs are defined in Table 1. From the involutionary view of classifications, the instance hypergraph is dual to the type hypergraph. The instance hypergraph consists of

1.   a set of nodes (entity instances) $node(inst(A)) = inst(ent(A)) = universe(A)$,
2.   a set of edges (relation tuples) $edge(inst(A)) = inst(rel(A)) = tuple(A)$, and
3.   an *signature* function $\partial_0 : tuple(A) \to tuple(refer(A))$ that maps an edge $t \in tuple(A)$ to its vertex $\partial_0(t)$ of nodes.

Table 3: Hypergraph Component Definitions

| Hypergraph Component		Classification Component	Notion
$refer(inst(A))$	=	$inst(refer(A))$	*set pair*
$sign(inst(A))$	=	$inst(sign(A))$	*function*
$arity(inst(A))$	=	$inst(arity(A))$	*function*
$name(inst(A))$	=	$var(A)$	*set*
$node(inst(A))$	=	$univ(A)$	*set*
$edge(inst(A))$	=	$tuple(A)$	*set*
$refer(typ(A))$	=	$typ(refer(A))$	*function*
$sign(typ(A))$	=	$typ(sign(A))$	*function*
$arity(typ(A))$	=	$typ(arity(A))$	*function*
$var(typ(A))$	=	$var(A)$	*set*
$ent(typ(A))$	=	$typ(ent(A))$	*set*
$rel(typ(A))$	=	$typ(rel(A))$	*set*
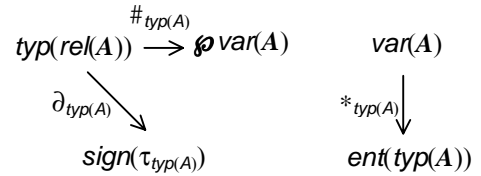
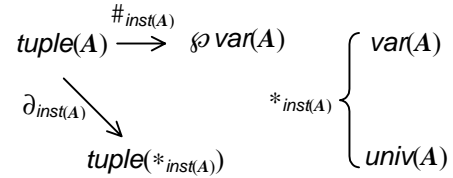**Figure 4: Type Language**

Figure 5: Instance Hypergraph

```
(12) (SET.FTN$function instance)
     (= (SET.FTN$source instance) model)
     (= (SET.FTN$target instance) hgph$hypergraph)
     (= (SET.FTN$composition [instance hgph$reference])
        (SET.FTN$composition [reference cls.sdsgn$instance]))
     (= (SET.FTN$composition [instance hgph$signature]
        (SET.FTN$composition [signature cls.dsgn$instance]))
     (= (SET.FTN$composition [instance hgph$arity])
        (SET.FTN$composition [arity cls.dsgn$instance]))
     (= (SET.FTN$composition [instance hgph$node]) universe)
     (= (SET.FTN$composition [instance hgph$edge]) tuple)
     (= (SET.FTN$composition [instance hgph$index]) variable)

(13) (SET.FTN$function type)
     (= (SET.FTN$source type) model)
     (= (SET.FTN$target type) lang$language)
     (= (SET.FTN$composition [type lang$reference])
        (SET.FTN$composition [reference cls.sdsgn$type]))
     (= (SET.FTN$composition [type lang$signature]
        (SET.FTN$composition [signature cls.dsgn$type]))
     (= (SET.FTN$composition [type lang$arity])
        (SET.FTN$composition [arity cls.dsgn$type]))
```

```
(= (SET.FTN$composition [type lang$entity])
   (SET.FTN$composition [entity cls$type]))
(= (SET.FTN$composition [type lang$relation])
   (SET.FTN$composition [relation cls$type]))
(= (SET.FTN$composition [type lang$variable]) variable)
```
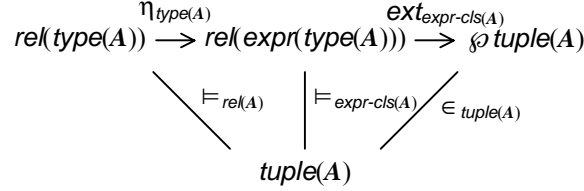
$$rel(type(A)) \xrightarrow{\eta_{type(A)}} rel(expr(type(A))) \xrightarrow{ext_{expr\text{-}cls(A)}} \wp\, tuple(A)$$

$$\vDash_{rel(A)} \qquad \vDash_{expr\text{-}cls(A)} \quad \in_{tuple(A)}$$

$$tuple(A)$$

Diagram 3: Expression Classification & Extension Function

$$\wp\, var(A) \xleftarrow{\#_{expr(L)} = arity(expr(type(A)))} rel(expr(type(A))) \xrightarrow{ext_{expr\text{-}cls(A)}} \wp\, tuple(A)$$

$$\vDash_{sup(var(A))} \qquad\qquad \vDash_{expr\text{-}cls(A)} \quad \in_{tuple(A)}$$

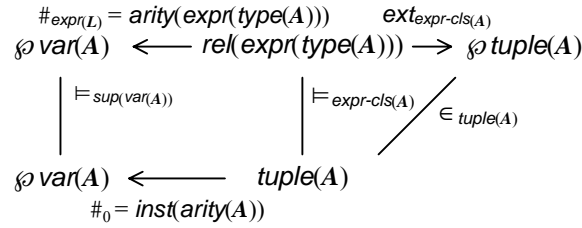$$\wp\, var(A) \xleftarrow{\#_0 = inst(arity(A))} tuple(A)$$

Diagram 4: Arity & Extension Functions

○   For any model A, we extend the relation classification

$$rel(A) = \langle rel(type(A)) = typ(rel(A)),\ tuple(A),\ \vDash_{rel(A)}\rangle$$

to an *expression* classification

$$expr\text{-}cls(A) = \langle rel(expr(type(A))),\ tuple(A),\ \vDash_{expr\text{-}cls(A)}\rangle.$$

We do this in an indirect manner by first defining the *extent* set-valued function

$$ext(expr\text{-}cls(A)) : rel(expr(type(A))) \to \wp\, tuple(A).$$

We can then define the expression classification to be the classification induced by this set-valued function.

There is a precursor function

$$cmpt(A) : rel(expr(type(A))) \to \wp\, tuple(A)$$

that defines when a tuple $t \in tuple(A)$ is compatible with an expression $\varphi \in rel(expr(type(A)))$:

$$t \in cmpt(A)(\varphi) \ \underline{\textbf{means}}\ inst(arity(A))(t) \supseteq arity(expr(L))(\varphi).$$

Any tuple classifiable by φ is also compatible with φ: $ext(expr\text{-}cls(A))(\varphi) \subseteq cmpt(A)(\varphi)$. Of course, although a tuple that is compatible with φ may not be classifiable by φ.

```
(14) (SET.FTN$function set-pair)
     (= (SET.FTN$source set-pair) model)
     (= (SET.FTN$target set-pair) set.pr$pair)
     (= (SET.FTN$composition [set-pair set.pr$set1])
        (SET.FTN$composition [type lang.expr$set]))
     (= (SET.FTN$composition [set-pair set.pr$set2]) tuple)

(15) (SET.FTN$function function-pair)
     (= (SET.FTN$source function-pair) model)
     (= (SET.FTN$target function-pair) set.ftn.pr$pair)
     (= (SET.FTN$composition [function-pair set.ftn.pr$source]) set-pair)
     (= (SET.FTN$composition [function-pair set.ftn.pr$target])
        (SET.FTN$composition [variable set$pair]))
     (= (SET.FTN$composition [function-pair lang.ftn.pr$function1]))
```

```
            (SET.FTN$composition [type lang.expr$arity]))
        (= (SET.FTN$composition [function-pair lang.ftn.pr$function2])
            (SET.FTN$composition [instance hgph$arity]))

(16) (SET.FTN$function induction)
        (= (SET.FTN$source induction) model)
        (= (SET.FTN$target induction) cls$classification)
        (= (SET.FTN$composition [induction cls$instance]) tuple)
        (= (SET.FTN$composition [induction cls$type])
            (SET.FTN$composition [type lang.expr$set]))
        (forall (?a (model ?a))
            (= (induction ?a)
                (set.ftn.pr$source
                    (set.ftn.pr$isotaxy
                        [(function-pair ?a) (set$super (variable ?a))]))))

(17) (SET.FTN$function compatible)
        (= (SET.FTN$source compatible) model)
        (= (SET.FTN$target compatible) set.ftn$function)
        (= (SET.FTN$composition [compatible set.ftn$source])
            (SET.FTN$composition [type lang.expr$set]))
        (= (SET.FTN$composition [compatible set.ftn$target])
            (SET.FTN$composition [tuple set$power]))
        (= compatible (SET.FTN$composition [induction cls$extent]))
```

○   There is also a strict compatible function

$$\bullet cmpt(A) : rel(expr(type(A))) \to \wp \, tuple(A)$$

where $t \in \bullet cmpt(A)(\varphi)$ __means__ $inst(arity(A))(t) = arity(expr(L))(\varphi)$.

```
(18) (SET.FTN$function strict-compatible)
        (= (SET.FTN$source strict-compatible) model)
        (= (SET.FTN$target strict-compatible) set.ftn$function)
        (= (SET.FTN$composition [strict-compatible set.ftn$source])
            (SET.FTN$composition [type lang.expr$set]))
        (= (SET.FTN$composition [strict-compatible set.ftn$target])
            (SET.FTN$composition [tuple set$power]))
        (forall (?a (model ?a)
                 ?t ((tuple ?a) ?t)
                 ?e ((lang.expr$set (type ?a)) ?e))
            (<=> (((strict-compatible ?a) ?e) ?t)
                (= ((lang.expr$arity (type?a)) ?e)
                    ((hgph$edge-arity (instance ?a)) ?t))))
```

○   For any expression $\varphi \in rel(expr(type(A)))$ there is a surjective, idempotent trim function from the set
of compatible tuples $cmpt(A)(\varphi)$ to the set of strictly compatible tuples $\bullet cmpt(A)(\varphi)$

$$trim(A)(\varphi) : cmpt(A)(\varphi) \to \bullet cmpt(A)(\varphi).$$

This function trims each tuple $t \in cmpt(A)(\varphi)$ returning only the part of t that is essential in the classi-
fication $t \vDash_{expr\text{-}cls(A)} \varphi$. Thus, $trim(A)(\varphi)$ restricts t to the subset $arity(\varphi) \subseteq arity(t)$.

```
(19) (KIF$function trim)
        (= (KIF$source trim) model)
        (= (KIF$target trim) SET.FTN$function)
        (forall (?a (model ?a))
            (and (= (SET.FTN$source (trim ?a)) (lang.expr$arity (type?a)))
                 (= (SET.FTN$target (trim ?a)) set.ftn$function)
                 (= (SET.FTN$composition [(trim ?a) set.ftn$target])
                    (ccompatable ?a))
                 (= (SET.FTN$composition [(trim ?a) set.ftn$target])
                    (strict-compatable ?a))
                 (forall (?e ((lang.expr$set (type ?a)) ?e)
                          ?t (((compatible ?a) ?e) ?t))
                    (= (((trim ?a) ?e) ?t)
                        (set.ftn$composition
                            [(set.ftn$inclusion []) t]
                                [(((lang.expr$arity (type?a)) ?e)
                                    ((hgph$edge-arity (instance ?a)) ?t)])
                            t))))))
```

○ The extent function is defined as the cotupling of an *extent tuple* of functions. The functions in this tuple are defined recursively. Some need to use the compatible function in their definition. Here are the definitions of the component functions in the extent tuple.

– [*primitive extent*]
Let $\rho \in$ *rel*(*type*(*A*)) be any relation type (primitive expression) and let $t \in$ *tuple*(*A*) be any tuple.
$t \in$ *ext*$_{expr\text{-}cls(A)}(\rho)$ iff $t \in$ *ext*$_{rel(A)}(\rho)$.

```
(20) (SET.FTN$function primitive-extent)
     (= (SET.FTN$source primitive-extent) model)
     (= (SET.FTN$target primitive-extent) set.ftn$function)
     (= (SET.FTN$composition [primitive-extent set.ftn$source])
        (SET.FTN$composition [type lang$relation]))
     (= (SET.FTN$composition [primitive-extent set.ftn$target])
        (SET.FTN$composition [tuple set$power]))
     (= primitive-extent (SET.FTN$composition [relation cls$extent])))
```

– [*negation extent*]
Let $\varphi \in$ *rel*(*expr*(*type*(*A*))) be any expression and let $t \in$ *tuple*(*A*) be any tuple.
$t \in$ *ext*$_{expr\text{-}cls(A)}(\neg\varphi)$ iff $t \in$ *cmpt*(*A*)$(\neg\varphi)$ and $t \notin$ *ext*$_{expr\text{-}cls(A)}(\varphi)$.

```
(21) (SET.FTN$function negation-extent)
     (= (SET.FTN$source negation-extent) model)
     (= (SET.FTN$target negation-extent) set.ftn$function)
     (= (SET.FTN$composition [negation-extent set.ftn$source])
        (SET.FTN$composition [type lang.expr$set]))
     (= (SET.FTN$composition [negation-extent set.ftn$target])
        (SET.FTN$composition [tuple set$power]))
     (forall (?a (model ?a)
             ?e ((lang.expr$set (type ?a)) ?e))
        (= ((negation-extent ?a) ?e)
           (set$binary-intersection
               [((compatible ?a) ?e) (set$complement ((extent ?a) ?e))]))))
```

Table 4: Connective operations

$\wedge \quad \mapsto \quad \cap_{tuple(A)} : \wp\, tuple(A) \times \wp\, tuple(A) \to \wp\, tuple(A)$

$\vee \quad \mapsto \quad \cup_{tuple(A)} : \wp\, tuple(A) \times \wp\, tuple(A) \to \wp\, tuple(A)$

$\Rightarrow \quad \mapsto \quad \Rightarrow_{tuple(A)} : \wp\, tuple(A) \times \wp\, tuple(A) \to \wp\, tuple(A)$

$\Leftrightarrow \quad \mapsto \quad \Leftrightarrow_{tuple(A)} : \wp\, tuple(A) \times \wp\, tuple(A) \to \wp\, tuple(A)$

Let $\varphi, \psi \in$ *rel*(*expr*(*type*(*A*))) be any two expressions and let $t \in$ *tuple*(*A*) be any tuple.

[*conjunction extent*]
$t \in$ *ext*$_{expr\text{-}cls(A)}(\varphi \wedge \psi)$ iff $[t \in$ *ext*$_{expr\text{-}cls(A)}(\varphi)$ <u>and</u> $t \in$ *ext*$_{expr\text{-}cls(A)}(\psi)]$.

[*disjunction extent*]
$t \in$ *ext*$_{expr\text{-}cls(A)}(\varphi \vee \psi)$ iff $t \in$ *cmpt*(*A*)$(\varphi \vee \psi)$ and $[t \in$ *ext*$_{expr\text{-}cls(A)}(\varphi)$ <u>or</u> $t \in$ *ext*$_{expr\text{-}cls(A)}(\psi)]$.

[*implication extent*]
$t \in$ *ext*$_{expr\text{-}cls(A)}(\varphi \Rightarrow \psi)$ iff $t \in$ *cmpt*(*A*)$(\varphi \Rightarrow \psi)$ and $[t \in$ *ext*$_{expr\text{-}cls(A)}(\varphi)$ <u>implies</u> $t \in$ *ext*$_{expr\text{-}cls(A)}(\psi)]$.

[*equivalence extent*]
$t \in$ *ext*$_{expr\text{-}cls(A)}(\varphi \Leftrightarrow \psi)$ iff $t \in$ *cmpt*(*A*)$(\varphi \Leftrightarrow \psi)$ and $[t \in$ *ext*$_{expr\text{-}cls(A)}(\varphi)$ <u>iff</u> $t \in$ *ext*$_{expr\text{-}cls(A)}(\psi)]$.

```
(22) (SET.FTN$function connective-operation)
     (= (SET.FTN$source connective-operation) lang.expr$connective)
     (= (SET.FTN$target connective-operation) set.ftn$function)
     (= (SET.FTN$composition [connective-operation set.ftn$source])
        (SET.FTN$composition
            [(SET.FTN$composition [tuple set$power]) set.lim.prd$square]))
     (= (SET.FTN$composition [connective-operation set.ftn$target])
        (SET.FTN$composition [tuple set$power]))
     (forall (?k (lang.expr$connective ?k)
             ?x1 ((set$power (tuple ?a)) ?x1)
             ?x2 ((set$power (tuple ?a)) ?x2))
        (and (=> (= ?k conjunctive)
```

```
                    (= ((connective-operation ?k) [?x1 ?x2])
                       (set$binary-intersection [?x1 ?x2])))
              (=> (= ?k disjunctive)
                    (= ((connective-operation ?k) [?x1 ?x2])
                       (set$binary-union [?x1 ?x2])))
              (=> (= ?k implicative)
                    (= ((connective-operation ?k) [?x1 ?x2])
                       (set$implication [?x1 ?x2])))
              (=> (= ?k equivalent)
                    (= ((connective-operation ?k) [?x1 ?x2])
                       (set$equivalence [?x1 ?x2]))))))

(23) (KIF$function connective-extent)
     (= (KIF$source connective-extent) model)
     (= (KIF$target connective-extent) SET.FTN$function)
     (forall (?a (model ?a))
         (and (= (SET.FTN$source (connective-extent ?a)) lang.expr$connective)
              (= (SET.FTN$target (connective-extent ?a)) set.ftn$function)
              (= (SET.FTN$composition [(connective-extent ?a) set.ftn$target])
                 ((SET.FTN$constant [lang.expr$connective set$set])
                    (set$power (tuple ?a))))
              (forall (?k (lang.expr$connective ?k)
                       ?e1 ((lang.expr$set (type ?a)) ?e1)
                       ?e2 ((lang.expr$set (type ?a)) ?e2))
                 (= (((connective-extent ?a) ?k) [?e1 ?e2])
                    (set$binary-intersection
                      [((compatible ?a)
                          (((lang.expr$injection (type ?a)) ?k) [?e1 ?e2]))
                       ((connective-operation ?k)
                          [((extent ?a) ?e1) ((extent ?a) ?e2)])])))))
```
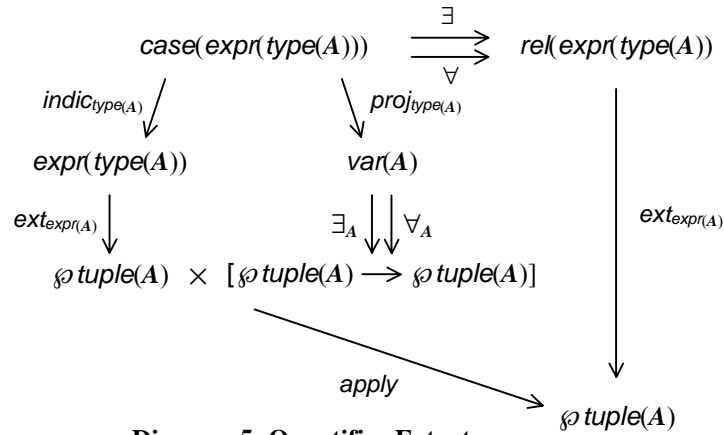

Diagram 5: Quantifier Extent

– Let $\varphi \in rel(expr(type(A)))$ be any expression, let $x \in arity(expr(L))(\varphi)$ be any variable in its arity, and let $t \in tuple(A)$ be any tuple. Recall the projection function

$$\pi_A(x) : tuple(A) \to tuple(A)$$

and the existential and universal functions

$$\exists_{A,x} = \exists \pi_A(x) : \wp\, tuple(A) \to \wp\, tuple(A) \text{ and } \forall_{A,x} = \forall \pi_A(x) : \wp\, tuple(A) \to \wp\, tuple(A),$$

where

$$\exists_{A,x}(X) = \{t \in tuple(A) \mid \exists s \in tuple(A) \text{ such that } \pi_A(x)(s) = t \text{ and } s \in X\}$$

$$\forall_{A,x}(X) = \{t \in tuple(A) \mid \forall s \in tuple(A) \text{ such that if } \pi_A(x)(s) = t \text{ then } s \in X\}$$

for any subset of tuples $X \in \wp\, tuple(A)$.

[*existential quantification extent*]

$t \in ext_{expr\text{-}cls(A)}((\exists x)\varphi)$ iff

$\exists s \in tuple(A)$ such that $\pi_A(x)(s) = t$ and $s \in ext_{expr\text{-}cls(A)}(\varphi)$.

So that, $ext_{expr\text{-}cls(A)}((\exists x)\varphi) = \exists_{A,\,x}(ext_{expr\text{-}cls(A)}(\varphi))$ – the application of a function to a source element. Clearly, if $s \in cmpt(A)(\varphi)$ then $\pi_A(x)(s) \in cmpt(A)((\exists x)\varphi)$.

[*universal quantification extent*]

$t \in ext_{expr\text{-}cls(A)}((\forall x)\varphi)$ iff

$\forall s \in tuple(A)$ if $\pi_A(x)(s) = t$ then $s \in ext_{expr\text{-}cls(A)}(\varphi)$.

So that, $ext_{expr\text{-}cls(A)}((\forall x)\varphi) = \forall_{A,\,x}(ext_{expr\text{-}cls(A)}(\varphi))$.

```
(24) (SET.FTN$function quantifier-operation)
     (= (SET.FTN$source quantifier-operation) lang.expr$quantifier)
     (= (SET.FTN$target quantifier-operation) set.ftn$function)
     (= (SET.FTN$composition [quantifier-operation set.ftn$source]) variable)
        ((SET.FTN$constant [lang.expr$quantifier set$set]) (variable ?a)))
     (= (SET.FTN$composition [quantifier-operation set.ftn$target])
        ((SET.FTN$constant [lang.expr$quantifier set$set]) (set$power (tuple ?a))))
     (forall (?a (model ?a)
              ?k (lang.expr$quantifier ?k))
        (and (=> (= ?k lang.expr$existential)
                 (= ((quantifier-operation ?a) ?k)
                    (set.ftn$exists (projection ?a))))
             (=> (= ?k lang.expr$universal)
                 (= ((quantifier-operation ?a) ?k)
                    (set.ftn$forall (projection ?a)))))))

(25) (KIF$function quantifier-extent)
     (= (KIF$source quantifier-extent) model)
     (= (KIF$target quantifier-extent) SET.FTN$function)
     (forall (?a (model ?a))
        (and (= (SET.FTN$source (quantifier-extent ?a)) lang.expr$quantifier)
             (= (SET.FTN$target (quantifier-extent ?a)) set.ftn$function)
             (= (SET.FTN$composition [(quantifier-extent ?a) set.ftn$source])
                ((SET.FTN$constant [lang.expr$quantifier set$set])
                    (lang$case (type (lang.expr$expression ?a)))))
             (= (SET.FTN$composition [(quantifier-extent ?a) set.ftn$target])
                ((SET.FTN$constant [lang.expr$quantifier set$set])
                    (set$power (tuple ?a))))
             (forall (?k (lang.expr$quantifier ?k))
                (= ((quantifier-extent ?a) ?k)
                   (set.ftn$composition
                       [(set.lim.prd2$pairing
                           [(set.ftn$composition
                               [(lang$indication (type ?a)) (extent ?a)])
                            (set.ftn$composition
                               [(lang$projection (type ?a))
                                ((quantifier-operation ?a) ?k)])])
                        (set.ftn$apply
                           [(set$power (tuple ?a)) (set$power (typle ?a))])])))))))
```

- [*substitution extent*]

    Let (φ, h) be a substitutable pair, consisting of an expression $\varphi \in rel(expr(type(A)))$ and a substitution $h : dom(type(A))(h) \to cod(type(A))(h)$. Substitutable means that the arity of φ matches the domain of h

    $$dom(type(A))(h) = arity(expr(type(A)))(\varphi).$$

    There is a surjective, idempotent trim operation

    $$trim(A)(\varphi) : cmpt(A)(\varphi) \to \bullet cmpt(A)(\varphi)$$

    from the compatible tuples of any expression $\varphi \in rel(expr(type(A)))$ to the strictly compatible tuples. Clearly, we have the identity

    $$t \in cmpt(A)(\varphi) \text{ iff } trim(A)(\varphi)(t) \in \bullet cmpt(A)(\varphi).$$

    The strict tuple substitution function

    $$\bullet subst(A)((\varphi, h)) : \bullet cmpt(A)(\varphi[h]) \to tuple(A)$$

is define by composition: $t \mapsto h \cdot t$

We define the strict extent of a substitution expression as follows:

$$t \in \textit{ext}_{\bullet expr\text{-}cls(A)}(\varphi[h]) \text{ iff } h \cdot t \in \textit{ext}_{expr\text{-}cls(A)}(\varphi).$$

The full extent is then defined as:

$$t \in \textit{ext}_{expr\text{-}cls(A)}(\varphi[h]) \text{ iff } h \cdot \textit{trim}(A)(\varphi[h])(t) \in \textit{ext}_{expr\text{-}cls(A)}(\varphi).$$

$$\text{iff } \textit{subst}(A)((\varphi, h))(\textit{trim}(A)(\varphi[h])(t)) \in \textit{ext}_{expr\text{-}cls(A)}(\varphi).$$

Using the trim and inclusion functions, define the (full) tuple substitution function

$$\textit{subst}(A)((\varphi, h)) = \textit{trim}(A)(\varphi[h]) \cdot \bullet\textit{subst}(A)((\varphi, h)) : \textit{cmpt}(A)(\varphi) \to \textit{tuple}(A)$$

Then

$$t \in \textit{ext}_{expr\text{-}cls(A)}(\varphi[h]) \text{ iff } \textit{subst}(A)((\varphi, h))(t) \in \textit{ext}_{expr\text{-}cls(A)}(\varphi).$$

So that, $\textit{ext}_{expr\text{-}cls(A)}(\varphi[h]) = \textit{subst}(A)((\varphi, h))^{-1}(\textit{ext}_{expr\text{-}cls(A)}(\varphi)).$

```
(26) (KIF$function strict-substitution)
     (= (KIF$source strict-substitution) model)
     (= (KIF$target strict-substitution) SET.FTN$function)
     (forall (?a (model ?a))
         (and (= (SET.FTN$source (strict-substitution ?a))
                 (rel$extent (lang$substitutable (lang.expr$expression (type ?a)))))
              (= (SET.FTN$target (strict-substitution ?a)) set.ftn$function)
              (forall (?e ?h ((lang$substitutable (lang.expr$expression (type ?a)) ?e ?h))
                  (and (= (set.ftn$source ((strict-substitution ?a) [?e ?h]))
                          ((strict-compatible ?a)
                              (((lang.expr$injection (type ?a)) substitutable) [?e ?h])))
                       (= (set.ftn$target ((strict-substitution ?a) [?e ?h]))
                          (tuple ?a))
                       (forall (?t ((strict-compatible ?a)
                          (((lang.expr$injection (type ?a)) substitutable) [?e ?h])) ?t))
                          (= (((strict-substitution ?a) [?e ?h]) ?t)
                             (set.ftn$composition [?h ?t]))))))

(27) (KIF$function substitution)
     (= (KIF$source substitution) model)
     (= (KIF$target substitution) SET.FTN$function)
     (forall (?a (model ?a))
         (and (= (SET.FTN$source (substitution ?a))
                 (rel$extent (lang$substitutable (lang.expr$expression (type ?a)))))
              (= (SET.FTN$target (substitution ?a)) set.ftn$function)
              (forall (?e ?h ((lang$substitutable (lang.expr$expression (type ?a)) ?e ?h))
                  (and (= (set.ftn$source ((substitution ?a) [?e ?h]))
                          ((compatible ?a)
                              (((lang.expr$injection (type ?a)) substitutable) [?e ?h])))
                       (= (set.ftn$target ((substitution ?a) [?e ?h]))
                          (tuple ?a))
                       (= ((substitution ?a) [?e ?h])
                          (set.ftn$composition
                              [((trim ?a) (((lang.expr$injection (type ?a)) substitutable) [?e ?h]))
                               ((strict-substitution ?a) [?e ?h])])))))))

(28) (SET.FTN$function substitution-extent)
     (= (SET.FTN$source substitution-extent) model)
     (= (SET.FTN$target substitution-extent) set.ftn$function)
     (forall (?a (model ?a)
         (and (= (set.ftn$source (substitution-extent ?a))
                 (rel$extent (lang$substitutable (lang.expr$expression (type ?a)))))
              (= (set.ftn$target (substitution-extent ?a))
                 (set$power (tuple ?a)))
              (forall (?e ?h ((lang$substitutable (lang.expr$expression (type ?a)) ?e ?h))
                  (= ((substitution-extent ?a) [?e ?h])
                     ((set.ftn$inverse-image ((substitution ?a) [?e ?h])) ((extent ?a) ?e)))))))
```

○ As stated above, for any model $A = \langle refer(A), sign(A) \rangle$ the *extent* function

$$extent(A) : set(typ(A)) \to \wp \, tuple(A)$$

is defined as the cotupling of an *extent tuple* of functions. The *expression classification*

$$expr\text{-}cls(A) = \langle tuple(A), set(typ(A)), \vDash \rangle$$

is the unique classification associated with this set-valued function – its induction.

```
(29) (KIF$function extent-tuple)
     (= (KIF$source extent-tuple) model)
     (= (KIF$target extent-tuple) SET.FTN$function)
     (forall (?a (model ?a))
          (and (= (SET.FTN$source (extent-tuple ?a)) lang.expr$kind)
               (= (SET.FTN$target (extent-tuple ?a)) set.ftn$function)
               (= (SET.FTN$composition [(extent-tuple ?a) set.ftn$target])
                 ((SET.FTN$constant [lang.expr$kind set$set]) (set$power (tuple ?a))))
               (= ((extent-tuple ?a) lang.expr$primitive) (primitive-extent ?a))
               (= ((extent-tuple ?a) lang.expr$negative) (negation-extent ?a))
               (forall (?k (lang.expr$connective ?k))
                   (= ((extent-tuple ?a) ?k) ((connective-extent ?a) ?k)))
               (forall (?k (lang.expr$quantifier ?k))
                   (= ((extent-tuple ?a) ?k) ((quantifier-extent ?a) ?k)))
              (= ((extent-tuple ?a) lang.expr$substitutive) (substitution-extent ?a))))

(30) (SET.FTN$function extent)
     (= (SET.FTN$source extent) model)
     (= (SET.FTN$target extent) set.ftn$set-valued)
     (= (SET.FTN$composition [extent set.ftn$source])
        (SET.FTN$composition [type lang.expr$set]))
     (= (SET.FTN$composition [extent set.ftn$base]) tuple)
     (forall (?a (model ?a))
         (= (extent ?a)
            ((set.col.coprd$cotupling (tuple-set (type ?a))) (extent-tuple ?a))))

(31) (SET.FTN$function expression-classification)
     (= (SET.FTN$source expression-classification) model)
     (= (SET.FTN$target expression-classification) cls$classification)
     (= expression-classification
        (SET.FTN$composition [extent set.ftn$induction]))
```

○ Any model $A = \langle refer(A), sign(A) \rangle$ can be extended to an *expression* model *expr*(*A*), whose reference semidesignation is the same, but whose relation classification is the expression classification for *A*.

<div style="float:right">

expr
model ⟶ model

type ↓   ↓ type

language ⟶ language
expr

Diagram 6: Commutation of expression with type

</div>

– $refer(expr(A)) = refer(A)$

$$= \langle inst(refer(A)), typ(refer(A)) \rangle : var(A) \hookrightarrow ent(A),$$

– $sign(expr(A))$

$$= \langle inst(sign(A)), sign(expr(type(A))) \rangle : expr\text{-}cls(A) \rightrightarrows sign(refer(A)).$$

The type language of the expression model is the expression language of the type language: the model expression function commutes with the underlying type function and the language expression function (Diagram 6).
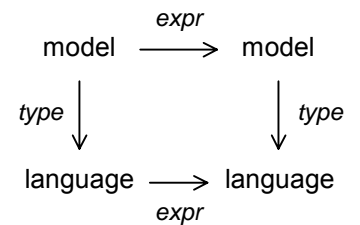
```
(32) (SET.FTN$function expression)
     (= (SET.FTN$source expression) model)
     (= (SET.FTN$target expression) model)
     (= (SET.FTN$composition [expression reference]) reference)
     (= (SET.FTN$composition [(SET.FTN$composition [expression signature]) cls.dsgn$source])
         expression-classification)
     (= (SET.FTN$composition [(SET.FTN$composition [expression signature]) cls.dsgn$target])
        (SET.FTN$composition [signature cls.dsgn$target]))
     (= (SET.FTN$composition [(SET.FTN$composition [expression signature]) cls.dsgn$instance])
        (SET.FTN$composition [signature cls.dsgn$instance]))
     (= (SET.FTN$composition [(SET.FTN$composition [expression signature]) cls.dsgn$type])
        (SET.FTN$composition [type lang.expr$signature]))
```

○ There is a simple *embedding* infomorphism from any model *A* to its expression model *expr*(*A*), whose relation infomorphism has identity instance function and primitive embedding type function.

$$embed(A) = \langle horiz\text{-}id(refer(A)), sign(embed(A)) \rangle : A \rightarrow expr(A),$$
where

$$\textit{vert-src}(\textit{sign}(\textit{embed}(A))) = \langle \textit{id}(\textit{tuple}(A)), \textit{atom}(A)\rangle : \textit{rel}(A) \rightleftharpoons \textit{expr-cls}(A).$$

```
(33) (SET.FTN$function embedding)
     (= (SET.FTN$source expression) model)
     (= (SET.FTN$target expression) mod.info$simple)
     (= (SET.FTN$composition [embedding source]) (SET.FTN$identity model))
     (= (SET.FTN$composition [embedding target]) expression)
     (= (SET.FTN$composition [(SET.FTN$composition [(SET.FTN$composition
           [embedding signature]) cls.sqr$vertical-source]) cls.info$instance]))
        (SET.FTN$composition [tuple set.ftn$identity]))
     (= (SET.FTN$composition [(SET.FTN$composition [(SET.FTN$composition
           [embedding signature]) cls.sqr$vertical-source]) cls.info$type]))
        atom)
```

Satisfaction

According to Chang and Keisler (Model Theory 1973), the notion of satisfaction is the cornerstone of model theory. Satisfaction refers to a binary relation between models over a type language L and sentences of that same language. An L-model $A \in mod(L)$ *satisfies* an L-sentence $\sigma \in sent(L)$,

$$A \vDash_L \sigma,$$

when σ is true when interpreted in the context A. Alternate terminology is shown in the following list (from Chang and Keisler).

> σ *holds in A*
> *A satisfies* σ
> σ *is satisfied in A*
> *A is a model of* σ

In this section we will define the notion of satisfaction and the related notion of representation. Since expressions are relation types for a suitable language, it does not matter whether we define these for relation types or expressions. If we define them for the relation types of an arbitrary model, then they have been defined for the expressions of any model A since they have been defined for the relation types of $expr(A)$. If we have defined them for all expression, then we have defined them for sentences, since sentences are expressions of empty arity.

Capsule argument.

Consider any relation type $\rho \in rel(typ(A))$. The type signature function

$$\partial_1 : typ(rel(A)) \rightarrow sign(typ(refer(A)))$$

maps ρ to $\partial_1(\rho) \in sign(typ(refer(A)))$, a signature of the reference function, which is a type in the signature classification of the reference semidesignation. The latter has an extent $ext_{sign(refer(A))}(\partial_1(\rho)) \subseteq tuple(refer(A))$ consisting of those concrete tuples $a \in tuple(refer(A))$ having $\partial_1(\rho)$ as their type. By definition of the signature classification $sign(refer(A))$ for the reference semidesignation, $a \in ext_{sign(refer(A))}(\partial_1(\rho))$ iff $arity(a) \supseteq arity(\partial_1(\rho))$ and $a(x) \vDash_{ent(A)} \partial_1(\rho)(x)$ for all variables $x \in arity(\partial_1(\rho))$. The set of all abstract tuples $r \in tuple(A)$ that could possibly be in the abstract extent of the relation type ρ is $\partial_0^{-1}(ext_{sign(refer(A))}(\partial_1(\rho))) \subseteq tuple(refer(A))$. This is the inverse image of the concrete extent along the instance signature function

$$\partial_0 : inst(rel(A)) \rightarrow tuple(refer(A))$$

This is to be compared with the actual extent $ext_{rel(A)}(\rho) \subseteq tuple(A)$ in the relation classification. Since the signature designation has an implication constraint, the following inclusion always holds

$$ext_{rel(A)}(\rho) \subseteq \partial_0^{-1}(ext_{sign(refer(A))}(\partial_1(\rho))).$$

We say that A *satisfies* ρ when this is an equality.

Extended discussion.

Initially, there are two points to make in our discussion of satisfaction. These points refer to the signature designation of a model

$$\partial_A = \langle \partial_0, \partial_1 \rangle = sign(A) : rel(A) \Rightarrow sign(refer(A)).$$

The models in the IFF Model Theory Ontology are an abstraction of the usual structures of model theory. In particular, tuples are an abstraction of the usual tuples and the extent of a relation type $\rho \in rel(typ(A))$, a subset of tuples, may not correspond exactly to the extent of the signature $\partial_1(\rho) \in sign(typ(refer(A)))$. This may occur for two reasons.

− The signature designation between the relation classification and the reference signature classification is expressed by an implication, not a logical equivalence:

$$r \vDash_{rel(A)} \rho \text{ implies } \partial_0(r) \vDash_{sign(refer(A))} \partial_1(\rho)$$

for tuple $r \in tuple(A)$ and relation type $\rho \in rel(typ(A))$.

So, a classification incidence $\partial_0(r) \vDash_{sign(refer(A))} \partial_1(\rho)$ may hold between the concrete representation of an abstract tuple and the signature of a relation type, but not between the abstract tuple and relation type themselves

$$r \nvDash_{rel(A)} \rho.$$

Representation can overcome this constraint.

- For the signature $\partial_1(\rho) \in sign(typ(refer(A)))$ of a relation type $\rho \in rel(typ(A))$ there may be a tuple classification $a \vDash_{sign(refer(A))} \partial_1(\rho)$ for some concrete tuple $a \in tuple(refer(A))$, where a is not the image of any abstract relational tuple $r \in tuple(A)$, much less the image of a tuple $r \in tuple(A)$ with

$$r \vDash_{rel(A)} \rho.$$

Satisfaction can overcome this constraint.

So for each relation type $\rho \in rel(typ(A))$, each concrete tuple $a \subseteq tuple(refer(A))$, and each concrete tuple classification $a \vDash_{sign(refer(A))} \partial_1(\rho)$, there are three possibilities.

1. There is an abstract tuple $r \in tuple(A)$ such that $r \vDash_{rel(A)} \rho$ and $a = \partial_0(r)$.

2. There is an abstract tuple $r \in tuple(A)$ such that $a = \partial_0(r)$, but $r \nvDash_{rel(A)} \rho$.

3. There is no abstract tuple $r \in tuple(A)$ with $r \vDash_{rel(A)} \rho$ and $a \le \partial_0(r)$.

- We say that a model A *represents* a relation type $\rho \in rel(typ(A))$ when possibility 2 does not hold; that is, when for all tuples $r \in tuple(A)$, $r \vDash_{rel(A)} \rho$ iff $\partial_0(r) \vDash_{sign(refer(A))} \partial_1(\rho)$.

- We say that a model A *satisfies* a relation type $\rho \in rel(typ(A))$ when possibility 3 does not hold; that is, when there is a tuple $r \in tuple(A)$ such that $r \vDash_{rel(A)} \rho$ and $a = \partial_0(r)$.

- We say that a structure A *strongly satisfies* a relation type $\rho \in rel(typ(A))$ when it both represents and satisfies ρ.

Encoding.

```
(34) (SET.FTN$function relationally-represents)
     (= (SET.FTN$source relationally-represents) model)
     (= (SET.FTN$target relationally-represents) set$set)
     (= relationally-represents (SET.FTN$composition [signature cls.dsgn$represents]))

(35) (SET.FTN$function relationally-satisfies)
     (= (SET.FTN$source relationally-satisfies) model)
     (= (SET.FTN$target relationally-satisfies) set$set)
     (= relationally-satisfies (SET.FTN$composition [signature cls.dsgn$satisfies]))

(36) (SET.FTN$function strongly-relationally-satisfies)
     (= (SET.FTN$source strongly-relationally-satisfies) model)
     (= (SET.FTN$target strongly-relationally-satisfies) set$set)
     (forall (?a (model ?a))
         (= (strongly-relationally-satisfies ?a)
            (set$binary-intersection
                [(relationally-represents ?a) (relationally-satisfies ?a)])))

(37) (SET.FTN$function expressively-represents)
     (= (SET.FTN$source expressively-represents) model)
     (= (SET.FTN$target expressively-represents) set$set)
     (= expressively-represents
        (SET.FTN$composition [expression relationally-represents]))

(38) (SET.FTN$function expressively-satisfies)
     (= (SET.FTN$source expressively-satisfies) model)
     (= (SET.FTN$target expressively-satisfies) set$set)
     (= expressively-satisfies
        (SET.FTN$composition [expression relationally-satisfies]))

(39) (SET.FTN$function strongly-expressively-satisfies)
     (= (SET.FTN$source strongly-expressively-satisfies) model)
```

```
        (= (SET.FTN$target strongly-expressively-satisfies) set$set)
        (= strongly-expressively-satisfies
           (SET.FTN$composition [expression strongly-relationally-satisfies]))

  (40) (SET.FTN$function satisfies)
        (= (SET.FTN$source satisfies) model)
        (= (SET.FTN$target satisfies) set$set)
        (forall (?a (model ?a))
            (= (satisfies ?a)
               (set$binary-intersection
                   [(lang$sentence (type ?a)) (expressively-satisfies ?a)])))
```

Review

Given any IFF language L, an IFF model (model-theoretic structure) A over language L, $typ(A) = L$, consists of a universe of entities $univ(A)$ and a tuple space $tuple(A)$. Both entities and tuples are untyped <u>a priori</u>. The model A types and interprets the entities and tuples via classifications – there is an entity classification $ent(A)$ whose instance set is the universe, and there is a relation classification $rel(A)$ whose instance set is the tuple space. These two classifications provide an interpretative semantics – an entity type $\varepsilon \in typ(ent(A))$ is interpreted as its extent $ext_{ent(A)}(\varepsilon) \subseteq univ(A)$ in the entity classification and a relation type $\rho \in typ(rel(A))$ is interpreted as its extent $ext_{rel(A)}(\rho) \subseteq tuple(A)$ in the relation classification. Any entity (instance) $a \in univ(A)$ is typed by an entity type $\varepsilon \in typ(ent(A))$ when it is classified by that type; that is, classification and typing are the same for entities. This is not true for relation – for relation classification is more general and freer than typing.

Any relation type $\rho \in typ(rel(A))$ has an arity $arity_{typ(A)}(\rho) \subseteq var(L)$ consisting of a set of variables, and has a signature consisting of a map (function) $\partial_{typ(A)}(\rho) : arity_{typ(A)}(\rho) \to typ(ent(A))$ from its arity to the set of entity types. Similarly, any tuple (relation instance) $t \in tuple(A)$ has an arity $arity_{inst(A)}(t) \subseteq var(L)$ consisting of a set of variables, and has a signature $\partial_{inst(A)}(t) : arity_{inst(A)}(\rho) \to univ(A)$ consisting of a map from its arity to the set of entity instances (elements of the universe). Relational typing and relational classification are codefinable. A tuple $t \in tuple(A)$ is typed by a relation type $\rho \in typ(rel(A))$ when the tuple $t \in tuple(A)$ is classified by the relation type $\rho \in typ(rel(A))$ and the arity of t is equal to the arity of ρ. A tuple $t \in tuple(A)$ is classified by a relation type $\rho \in typ(rel(A))$ when a subtuple of t is typed by the relation type $\rho \in typ(rel(A))$. A necessary condition in either case is that for each variable $x \in arity_{typ(A)}(\rho)$ that the entity $t(x)$ is classified (and hence, typed) by $\rho(x)$. Relational typing is a sub-classification of relation classification. Relational typing is the desired classification. However, relational classification is a useful technical device. In traditional semantics (*Model Theory* by Chang and Keisler), relational classification has been used to define satisfaction. In the IFF, relational classification, along with the abstractness of tuples, is used to show the existence of free models (and logics).

Any language L can be extended to a language $expr(L)$ whose components are those of L, except that relation types are expressions and relational classification is expression classification, the classification of tuples by expressions. Relational classification in L is inductively extended to an expression classification. A model A satisfies an expression φ when all tuples are classified by φ.

○   For any model A, rename the set of satisfied expressions as $thm(A)$, the theorems of A.

```
(41) (SET.FTN$function theorem)
     (= (SET.FTN$source theorem) model)
     (= (SET.FTN$target theorem) set$set)
     (forall (?a (model ?a))
         (set$subset (theorem ?a) (lang.expr$set (type ?a))))
     (= theorem expressively-satisfies)
```

○   For any model A, a theorem is an expression that holds for all tuples.

```
(forall (?a (model ?a))
        ?e ((relation (expression ?a)) ?e))
    (<=> ((theorem ?a) ?e)
        (forall (?t ((tuple ?a) ?t))
            ((expression-classification ?a) ?t ?e))))
```

○   The *theories* of a model $th(A)$ are all compatible theories satisfied by the model.

```
(42) (KIF$function theory)
     (= (KIF$source theory) model)
     (= (KIF$target theory) SET$class)
     (forall (?a (model ?a))
         (and (SET$subclass (theory ?a) th$theory)
             (forall (?t (th$theory ?t) (plog$compatible ?a ?t))
                 (<=> ((theory ?a) ?t)
                     (set$subset (th$axiom ?t) (theorem ?a))))))
```

○   For any model *A* and any abstract tuple *r* ∈ *tuple*(*A*), let *true*(*r*) ⊆ *typ*(*rel*(*expr*(*A*))) denote the set of all expressions that are satisfied by *r*. This is *intent*(*rel*(*expr*(*A*)))(*r*), the intent of *r* in the expression classification of *A*.

```
(43) (SET.FTN$function true)
     (= (SET.FTN$source true) model)
     (= (SET.FTN$target true) set.ftn$function)
     (= (SET.FTN$composition [true set.ftn$source]) tuple)
     (= (SET.FTN$composition [true set.ftn$target])
        (SET.FTN$composition [type lang.expr$set]))
     (forall (?a (model ?a))
         (= (true ?a)
            (cls$intent (expression-classification ?a))))
```

○   For any model *A* and any abstract tuple *r* ∈ *tuple*(*A*), all theorems of *A* are satisfied by *r*.

```
(44) (forall (?a (model ?a) ?r ((tuple ?a) ?r))
        (set$subset (theorem ?a) ((true ?a) ?r)))
```

○   Every model *A* generates a *maximal theory* *maxth*(*A*), whose underlying language is the type language of the model, and whose axioms are the theorems of the model.

```
(45) (SET.FTN$function maximal-theory)
     (= (SET.FTN$source maximal-theory) model)
     (= (SET.FTN$target maximal-theory) th$theory)
     (= (SET.FTN$composition [maximal-theory th$base]) type)
     (= (SET.FTN$composition [maximal-theory th$axiom]) theorem)
```

○   The maximal theory of a model is closed.

```
(46) (forall (?a (model ?a))
         (th$closed (maximal-theory ?a)))
```

○   Every theory of a model *A* is a sub-theory of the maximal theory *maxth*(*A*).

```
(47) (forall (?a (model ?a) ?t (th$theory ?t))
         (=> ((theory ?a) ?t) (th$subtheory ?t (maximal-theory ?a))))
```

○   Every model *A* generates a *logic* *log*(*A*), whose component model is *A* and whose theory *maxth*(*A*) is the maximal theory of *A*.

```
(48) (SET.FTN$function logic)
     (= (SET.FTN$source logic) model)
     (= (SET.FTN$target logic) log$logic)
     (= (SET.FTN$composition [logic log$model]) (SET.FTN$identity model))
     (= (SET.FTN$composition [logic log$theory]) maximal-theory)
```
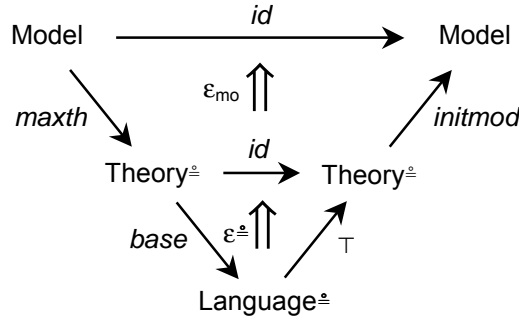
Language Intent Infomorphism

Figure 6: Counit of composite adjunction ω

○   [The counit of the composite $\omega = \kappa^{\triangleq\triangleright} \cdot \mu$]

$\langle \wp, typ, \text{id}, \varepsilon \rangle = \langle \top, base, \text{id}, \varepsilon^{\triangleq} \rangle \cdot \langle initmod, maxth, \eta, \varepsilon \rangle$

[A^{th} component of the counit of $\omega = \langle \wp, typ, \text{id}, \varepsilon \rangle : \text{Language}^{\triangleq} \to \text{Model}$]

Any classification A has an associated *intent infomorphism* (Figure 7)

$\varepsilon_A : \wp \, typ(A) \rightleftharpoons A$

that compares that classification A with the power classification of its type set. The intent infomorphism is the identity function on types and the intent function on instances. The fundamental property of the intent infomorphism is the definition of instance intent:

$t \in intent(A)(i) \text{ iff } i \models t$

for all instances $i \in inst(A)$ and all types $t \in typ(A)$.

In an analogous fashion, any model A has a *language intent* infomorphism

$\varepsilon_A : \wp \, typ(A) \rightleftharpoons A$

Figure 7: Intent Infomorphism on A

that compares that model A with the (anarchic) power model of its type language. The intent infomorphism ε_A is a two dimensional construction consisting of a reference classification quasisquare *refer*(ε_A) and a signature classification square *sign*(ε_A), where the signature infomorphism of the reference quasisquare is the vertical target of the signature square

$sign(refer(\varepsilon_A)) = vert\text{-}tgt(sign(\varepsilon_A)).$

The intent infomorphism is the identity language morphism on types and a modified edge-incident (intent) hypergraph morphism on instances. Its entity aspect (Figure 7a) is the intent infomorphism on the

Figure 7a: Intent Infomorphism on $ent(A)$

Figure 7b: Modified Intent Infomorphism on $rel(A)$

entity classification, and its relation aspect (Figure 7b) is a modified intent infomorphism on the relation classification. The instance function

$tuple(\varepsilon_A) : tuple(A) \to tuple(\wp\, typ(A))) = \tilde{\wp}\, rel(typ(A))$

maps a tuple $r \in tuple(A)$ to the tuple

$(arity\text{-}index(A)(r), intent(rel(A))(r))$.

It is straightforward to check that $rel(\varepsilon_A)$ is an infomorphism, satisfying the fundamental property:

$\rho \,\tilde{\in}\, tuple(\varepsilon_A)(r) \;\underline{\text{iff}}\; r \vDash_{rel(A)} \rho$

for all tuple $r \in tuple(A)$ and all relation types $\rho \in typ(rel(A))$.

```
(49) (SET.FTN$function language-intent-function)
     (= (SET.FTN$source language-intent-function) model)
     (= (SET.FTN$target language-intent-function) set.ftn$function)
     (= (SET.FTN$composition [language-intent-function set.ftn$source]) tuple)
     (= (SET.FTN$composition [language-intent-function set.ftn$target])
        (SET.FTN$composition [type lang$free-tuple]))
     (forall (?a (model ?a))
        (and (= (SET.FTN$composition
                   [(language-intent-function ?a) (lang$bounding-arity (type ?a))])
                (cls.dsgn$instance (arity a)))
             (= (SET.FTN$composition
                   [(language-intent-function ?a) (lang$subset (type ?a))])
                (cls$intent (relation a)))))

(50) (SET.FTN$function language-intent-infomorphism)
     (= (SET.FTN$source language-intent-infomorphism) model)
     (= (SET.FTN$target language-intent-infomorphism) cls.info$infomorphism)
     (= (SET.FTN$composition [language-intent-infomorphism cls.info$source])
        (SET.FTN$composition [type lang$free-relation]))
     (= (SET.FTN$composition [language-intent-infomorphism cls.info$target]) relation)
     (= (SET.FTN$composition [language-intent-infomorphism cls.info$type])
        (SET.FTN$composition [(SET.FTN$composition
           [relation cls$type]) set.ftn$identity]))
     (= (SET.FTN$composition [language-intent-infomorphism cls.info$instance])
        language-intent-function)

(51) (SET.FTN$function language-intent)
     (= (SET.FTN$source language-intent) model)
     (= (SET.FTN$target language-intent) mod.info$infomorphism)
     (= (SET.FTN$composition [language-intent mod.info$source])
        (SET.FTN$composition [type lang$power]))
     (= (SET.FTN$composition [language-intent mod.info$target])
        (SET.FTN$identity model))
     (= (SET.FTN$composition [language-intent mod.info$type])
        (SET.FTN$composition [type lang.mor$identity]))
     (= (SET.FTN$composition [language-intent mod.info$variable])
        (SET.FTN$composition [variable set.invpr$identity]))
     (= (SET.FTN$composition [language-intent mod.info$entity])
        (SET.FTN$composition [entity cls$intent]))
     (= (SET.FTN$composition [language-intent mod.info$relation])
        language-intent-infomorphism)
```

Finally, in order to check that ε_A is an infomorphism, the only remaining constraint to verify is the instance aspect of the signature square represented by the diagram in Figure 8. The function

$inst(sign(refer(\varepsilon_A))) : inst(sign(refer(A\,))) \to inst(sign(refer(\wp\, typ(A\,)))$

maps an instance tuple $r \in inst(sign(refer(A\,)))$ with arity $arity(r)$ to the coordinate-wise intents

$inst(sign(refer(\varepsilon_A)\,))(r)_x = intent(ent(A))(r_x)$ for any $x \in arity(r)$.

With the definition of a basic IFF model, the diagram in Figure 8 is not commutative. It only satisfies the inequality

$tuple(\varepsilon_A) \cdot inst(sign(\wp\, typ(A)))$

$\subseteq inst(sign(A)) \cdot inst(sign(refer(\varepsilon_A)))$.

The solution to this problem is to revisit the definition of a basic IFF model. In the notion of an IFF

Figure 8: Instance aspect of the signature square of the intent infomorphism

model, there are both unary relation types (predicates) and entity types (sorts). These are connected by the constraint that for any unary relation type $\rho \in typ(rel(A))$ whose index arity is (x) and whose signature is the one-tuple (ϵ) for some entity type $\epsilon \in typ(ent(A))$, for any tuple r, if $r \vDash_{rel(A)} \rho$ then $r_x \vDash_{rel(A)} \epsilon$. Put in more colloquial terms, if $\rho(e)$ holds, then e is of type (sort) ϵ. We need to define a notion of a constrained IFF model. The constrained IFF model (call it a *normal model* or some such name) further requires that

− Every entity type (sort) $\epsilon \in typ(ent(A))$ be the signature of a unary relation type (predicate) $\rho \in typ(rel(A))$, whose x^{th} projection is extensionally equivalent to it: $\rho(e)$ holds iff e is of sort ϵ.

− If a relation type $\rho \in typ(rel(A))$ is in the extent of a tuple $t \in tuple(A)$ and an entity type (sort) ϵ is a component in the signature of ρ, then the unary relation type associated with ϵ is in the extent of the tuple t.

With these assumptions, Figure 8 is a commutative diagram and the intent is an infomorphism.

○ This function is the counit

$$\epsilon = (maxth\ \epsilon^{\doteq}\ restrict) \cdot \epsilon_{mod}$$

of the composite adjunction $\omega = \kappa^{\doteq\triangleright} \cdot \mu$, where $\kappa^{\doteq\triangleright}$ has the counit $\epsilon^{\doteq}$ and μ has the counit ϵ_{mod}.

```
(52) (forall (?a (model ?a))
        (= (language-intent ?a)
           (mod.info$composition
              [(th.mor.spl$initial-model
                  (th$special-empty-inclusion
                      (maximal-theory ?a))
               (theory-intent ?a)]))))
```
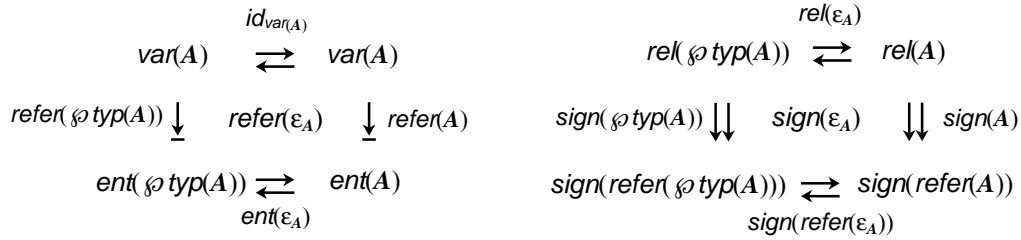

Figure 9a: Reference Classification Semisquare – abstract
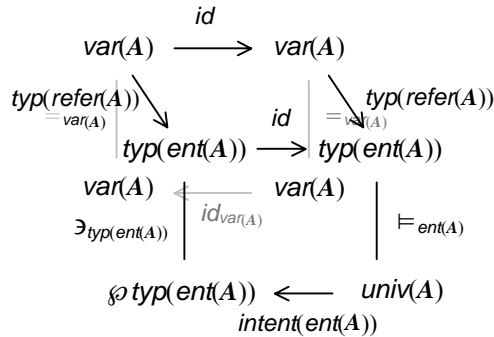
Figure 9b: Signature Classification Square – abstract
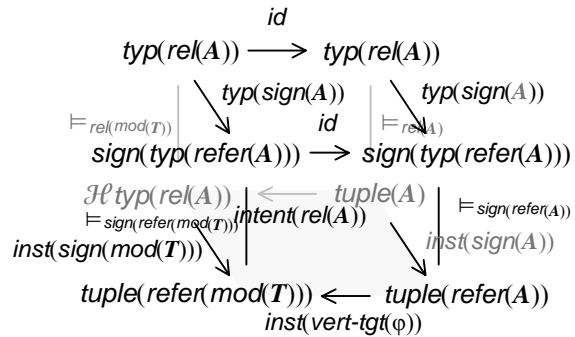

Figure 10a: Reference Classification Semisquare – details

Figure 10b: Signature Classification Square – details

Theory Intent Infomorphism

○ Consider the model *init-mod*(*max-th*(*A*)). This has the same underlying type language as *A*, including the same set of variables. And, the universe *univ*(*init-mod*(*max-th*(*A*))) = *univ*($\wp$ *typ*(*A*)) = $\wp$ *ent*(*typ*(*A*)) = $\wp$ *typ*(*ent*(*A*)) is the universe of the power of the underlying type language. However, by definition the set of *formal tuples*

$$\textit{fml-tuple}(A) \triangleq \textit{tuple}(\textit{init-mod}(\textit{maxth}(A))) \subseteq \textit{tuple}(\wp\,\textit{typ}(A)) = \textit{free-tuple}(\textit{typ}(A))$$

is the subset of all free tuples of *typ*(*A*) that satisfy all the theorems of *max-th*(*A*); for the special case *A* = $\wp$ *typ*(*A*) the set of formal tuples equals the set of free tuples. The following fact is used repeatedly throughout the discussions and axiomatization of the IFF Ontology (meta) Ontology.

For any abstract tuple $r \in$ *tuple*(*A*) the free tuple

"*intent*(*rel*(*A*))"(*r*) = (*arity*(*A*)(*r*), {ρ ∈ *rel*(*L*) | *r* ⊨$_{rel(A)}$ ρ}) ∈ *free-tuple*(*typ*(*A*)),

which is the relation intent of *r*, is a formal tuple of *A*; that is, it satisfies exactly the same relation types (and hence expressions) as *r*.

This means that the free relation intent function

"*intent*(*rel*(*A*))" : *tuple*(*A*) → *tuple*($\wp$ *typ*(*A*)) = *free-tuple*(*typ*(*A*))

factors through the set of formal tuples *fml-tuple*(*A*). Define the relation intent function, the edge function for the instance hypergraph morphism of the theory intent infomorphism, to be this co-restriction.

```
(53) (SET.FTN$function formal-tuple)
     (= (SET.FTN$source formal-tuple) model)
     (= (SET.FTN$source formal-tuple) set$set)
     (forall (?a (model ?a))
        (and (set$subset (formal-tuple ?a) (lang$free-tuple (type ?a))))
             (forall (?r ((lang$free-tuple (type ?a)) ?r))
                (<=> ((formal-tuple ?a) ?r)
                     (forall (?e ((theorem (maximal-theory ?a)) ?e))
                        ((expression-classification ?a) ?r ?e))))))

(54) (SET.FTN$function formal-relation)
     (= (SET.FTN$source formal-relation) language)
     (= (SET.FTN$target formal-relation) cls$classification)
     (= (SET.FTN$composition [formal-relation cls$type]) relation)
     (= (SET.FTN$composition [formal-relation cls$instance]) formal-tuple)
     (forall (?a (model ?a))
        (rel$abridgment (formal-relation ?a) (free-relation (type ?a))))

(55) (SET.FTN$function theory-intent-function)
     (= (SET.FTN$source theory-intent-function) model)
     (= (SET.FTN$target theory-intent-function) set.ftn$function)
     (= (SET.FTN$composition [theory-intent-function set.ftn$source]) tuple)
     (= (SET.FTN$composition [theory-intent-function set.ftn$target]) formal-tuple)
     (forall (?a (model ?a))
        (set.ftn$restriction
            (theory-intent-function ?a)
            (language-intent-function ?a)))

(56) (SET.FTN$function theory-intent-infomorphism)
     (= (SET.FTN$source theory-intent-infomorphism) model)
     (= (SET.FTN$target theory-intent-infomorphism) cls.info$infomorphism)
     (= (SET.FTN$composition [theory-intent-infomorphism cls.info$source])
        formal-relation)
     (= (SET.FTN$composition [theory-intent-infomorphism cls.info$target]) relation)
     (= (SET.FTN$composition [theory-intent-infomorphism cls.info$type])
        (SET.FTN$composition [(SET.FTN$composition
            [relation cls$type]) set.ftn$identity]))
     (= (SET.FTN$composition [theory-intent-infomorphism cls.info$instance])
        theory-intent-function)
```
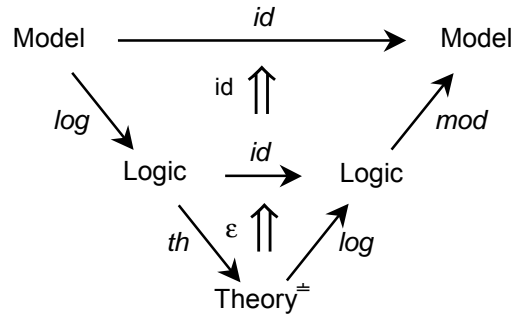
Figure 11: Counit of composite adjunction μ

○ [The counit of the composite μ = λ · π]

⟨*init-mod*, *max-th*, η, ε⟩ = ⟨*log*, *th*, id, ε⟩ · ⟨*mod*, *log*, η, id⟩

[A^{th} component of the counit of μ = ⟨*init-mod*, *max-th*, η, ε⟩ : Theory$^{\pm}$ → Model]

For every model *A*, there is an *theory intent* infomorphism

ε$_A$ = *intent*(*A*) : *init-mod*(*max-th*(*A*)) → *A*

whose type component is the identity special type language morphism and whose instance component is the intent hypergraph morphism, with entity intent as node function and modified and restricted relation intent as edge function.

```
(57) (SET.FTN$function theory-intent)
     (= (SET.FTN$source theory-intent) model)
     (= (SET.FTN$target theory-intent) mod.info$infomorphism)
     (= (SET.FTN$composition [theory-intent mod.info$source])
        (SET.FTN$composition [maximal-theory th$initial-model]))
     (= (SET.FTN$composition [theory-intent mod.info$target])
        (SET.FTN$identity model))
     (= (SET.FTN$composition [theory-intent mod.info$type])
        (SET.FTN$composition [type lang.mor.spl$identity]))
     (= (SET.FTN$composition [theory-intent mod.info$variable])
        (SET.FTN$composition [variable set.invpr$identity]))
     (= (SET.FTN$composition [theory-intent mod.info$entity])
        (SET.FTN$composition [entity cls$intent]))
     (= (SET.FTN$composition [theory-intent mod.info$relation])
        theory-intent-infomorphism)
```

○ This function is the counit

ε$_{\text{th}}$ = *log* ∘ ε$_{\text{log}}$ ∘ *mod*

of the composite adjunction μ = λ · π, where λ has the counit ε$_{\text{log}}$ and π has the identity counit.

```
(58) (= theory-intent
        (SET.FTN$composition [(SET.FTN$composition
           [logic log$intent]) log.info$model]))
```

○ For any model *A*, the *empty* prologic ⊤(*A*) has model component *A* and the empty theory over *typ*(*A*) as its theory component. This prologic is actually a logic, since no axioms need to be satisfied.

```
(59) (SET.FTN$function empty)
     (= (SET.FTN$source empty) model)
     (= (SET.FTN$target empty) plog$prologic)
     (= (SET.FTN$composition [empty plog$model])
        (SET.FTN$identity model))
     (= (SET.FTN$composition [empty plog$theory])
        (SET.FTN$composition [type lang$empty]))
```

○ For any model *A*, the *full* prologic ⊥(*A*) has model component *A* and the full theory over *typ*(*A*) as its theory component.

```
(60) (SET.FTN$function full)
     (= (SET.FTN$source full) model)
     (= (SET.FTN$target full) plog$prologic)
     (= (SET.FTN$composition [full plog$model])
        (SET.FTN$identity model))
     (= (SET.FTN$composition [full plog$theory])
        (SET.FTN$composition [type lang$full]))
```

Case and Spanmodels

○ There is a *case* classification $case(A) = \langle case(inst(A)), case(typ(A)), \models_{case(A)} \rangle$, whose instances (r, x) are tuple-variable pairs for tuples $r \in tuple(A) = edge(inst(A))$ and variables $x \in inst(arity(A))(r) \subseteq var(A)$, whose types (ρ, x) are relation-type-variable pairs for relation types $\rho \in typ(rel(A)) = rel(typ(A))$ and variables $x \in inst(arity(A))(\tau) \subseteq var(A)$, and whose classification is defined by

$(r, x) \models_{case(A)} (\rho, x')$ when $r \models_{rel(A)} \rho$ and $x = x'$.

```
(61) (SET.FTN$function case)
     (= (SET.FTN$source case) model)
     (= (SET.FTN$target case) cls$classification)
     (= (SET.FTN$composition [case cls$instance])
        (SET.FTN$composition [instance hgph$case]))
     (= (SET.FTN$composition [case cls$type])
        (SET.FTN$composition [type lang$case]))
     (forall (?a (model ?a)
             ?t ?x ((hgph$case (instance ?a)) [?t ?x])
             ?tau ?x1 ((lang$case (type ?a)) [?tau ?x1]))
        (<=> ((case ?a) [?t ?x] [?tau ?x1])
             (and ((relation ?a) ?t ?tau)
                  (= ?x ?x1))))
```

○ Any model $A = \langle refer(A), sign(A) \rangle$ has an associated *indication* designation

$indic(A)$

$= \langle indic(inst(A)), indic(typ(A)) \rangle : case(A) \Rightarrow rel(A),$

whose source is the case classification of A, whose target is the relation classification of A, whose instance function is the indication function for the instance hypergraph, and whose type function is indication function for the type language.

$$\begin{array}{ccc}
& indic(typ(A)) & \\
case(typ(A)) & \longrightarrow & typ(rel(A)) \\
\models_{case(A)} \Big| & indic(A) & \Big| \models_{rel(A)} \\
case(inst(A)) & \longrightarrow & tuple(A) \\
& indic(inst(A)) &
\end{array}$$

Figure 12: Indication designation

```
(62) (SET.FTN$function indication)
     (= (SET.FTN$source indication) model)
     (= (SET.FTN$target indication) cls.dsgn$designation)
     (= (SET.FTN$composition [indication cls.dsgn$source]) case)
     (= (SET.FTN$composition [indication cls.dsgn$target]) relation)
     (= (SET.FTN$composition [indication cls.dsgn$instance])
        (SET.FTN$composition [instance hgph$indication]))
     (= (SET.FTN$composition [indication cls.dsgn$type])
        (SET.FTN$composition [type lang$indication]))
```

○ Any model $A = \langle refer(A), sign(A) \rangle$ has an associated *projection* designation $proj(A)$

$= \langle proj(inst(A)), proj(typ(A)) \rangle : case(A) \Rightarrow cla(var(A)),$

whose source is the case classification of A, whose target is the identity classification of the variable set of A, whose instance function is the projection function for the instance hypergraph, and whose type function is projection function for the type language. The designation requirement that classification be preserved is evident from the definitions of the case classification, the projection for the instance hypergraph and the projection for the type language.

$$\begin{array}{ccc}
& proj(typ(A)) & \\
case(typ(A)) & \longrightarrow & var(A) \\
\models_{case(A)} \Big| & proj(A) & \Big| =_{var(A)} \\
case(inst(A)) & \longrightarrow & var(A) \\
& proj(inst(A)) &
\end{array}$$

Figure 13: Projection designation

```
(63) (SET.FTN$function projection)
     (= (SET.FTN$source projection) model)
     (= (SET.FTN$target projection) cls.dsgn$designation)
     (= (SET.FTN$composition [projection cls.dsgn$source]) case)
     (= (SET.FTN$composition [projection cls.dsgn$target])
        (SET.FTN$composition [variable set$classification]))
     (= (SET.FTN$composition [projection cls.dsgn$instance])
        (SET.FTN$composition [instance hgph$projection]))
     (= (SET.FTN$composition [projection cls.dsgn$type])
```

```
                 (SET.FTN$composition [type lang$projection]))
```

○   Any model $A = \langle refer(A), sign(A)\rangle$ has an associated *comediator* designation

$*_A = comed(A)$

$= \langle comed(inst(A)), comed(typ(A))\rangle : case(A) \Rightarrow ent(A),$

whose source is the case classification of A, whose target is the entity classification of A, whose instance function is the comediator function for the instance hypergraph, and whose type function is the comediator function for the type language.

$$comed(typ(A))$$
$$case(typ(A)) \longrightarrow typ(ent(A))$$

$$\vDash_{case(A)} \Big| \qquad comed(A) \qquad \Big| \vDash_{ent(A)}$$

$$case(inst(A)) \longrightarrow univ(A)$$
$$comed(inst(A))$$

Figure 14: Comediator designation

```
(64) (SET.FTN$function comediator)
     (= (SET.FTN$source comediator) model)
     (= (SET.FTN$target comediator) cls.dsgn$designation)
     (= (SET.FTN$composition [comediator cls.dsgn$source]) case)
     (= (SET.FTN$composition [comediator cls.dsgn$target]) entity)
     (= (SET.FTN$composition [comediator cls.dsgn$instance])
        (SET.FTN$composition [instance hgph$comediator]))
     (= (SET.FTN$composition [comediator cls.dsgn$type])
        (SET.FTN$composition [type lang$comediator]))
```

$$var(A) \qquad rel(A)$$

$$refer(A) \Big\downarrow \qquad sign(A) \Big\downarrow\Big\downarrow \qquad \Rightarrow$$

$$ent(A) \qquad sign(refer(A))$$

$$indic(A) \qquad comed(A)$$
$$rel(A) \;\Longleftarrow\; case(A) \;\Longrightarrow\; ent(A)$$

$$proj(A) \Big\downarrow\Big\downarrow$$

$$var(A)$$

Diagram 7a: Model to Spanmodel - abstract version

$$typ(p)$$
$$set(p) \longrightarrow typ(cla(p))$$

$$= \qquad \Big| \vDash_{cla(p)}$$

$$set(p) \qquad inst(cla(p))$$

$$typ(sign(A))$$
$$typ(rel(A)) \longrightarrow sign(typ(refer(A)))$$

$$\vDash_{rel(A)} \Big| \qquad \Big| \vDash_{sign(refer(A))}$$

$$tuple(A) \longrightarrow tuple(refer(A))$$
$$inst(sign(A))$$

$$\Rightarrow$$

$$indic(typ(A)) \qquad comed(typ(A))$$
$$typ(rel(A)) \longleftarrow case(typ(A)) \longrightarrow typ(ent(A))$$
$$proj(typ(A))$$
$$\vDash_{rel(A)} \Big| \quad indic(A) \;\vDash_{case(A)}\; comed(A) \qquad \Big| \vDash_{ent(A)}$$
$$var(A)$$
$$proj(A)$$
$$tuple(A) \longleftarrow case(inst(A)) \longrightarrow univ(A)$$
$$indic(inst(A)) \qquad comed(inst(A))$$
$$proj(inst(A)) \qquad \Big| =_{var(A)}$$
$$var(A)$$

Diagram 7b: Model to Spanmodel - detailed version

○   Associated with any model $A = \langle refer(A), sign(A)\rangle$ is a spanmodel $spnmod(A)$, whose vertex (prehension) classification is the case (role) classification of A,

–   whose first designation (actuality) is the comediator designation $1^{st}_{spnmod(A)} = comed(A)$ with target classification being the entity classification of A,

–   whose second designation (naming) is the projection designation $2^{nd}_{spnmod(A)} = proj(A)$ with target classification being the variable identity classification of A, and

–   whose third designation (nexus) is the indication designation $3^{rd}_{spnmod(A)} = indic(A)$ with target classification being the relation classification of A.

```
(65) (SET.FTN$function spanmodel)
     (= (SET.FTN$source spanmodel) model)
     (= (SET.FTN$target spanmodel) smod$spanmodel)
     (forall (?a (model ?a))
         (and (= (smod$vertex (spanmodel ?a)) (case ?a))
              (= (smod$first (spanmodel ?a)) (comediator ?a))
              (= (smod$classification1 (spanmodel ?a)) (entity ?a))
              (= (smod$second (spanmodel ?a)) (projection ?a))
              (= (smod$classification2 (spanmodel ?a)) (set$classification (variable ?a)))
              (= (smod$third (spanmodel ?a)) (indication ?a))
              (= (smod$classification3 (spanmodel ?a)) (relation ?a))))
```

Infomorphisms

`mod.info`

Models are connected by and comparable with infomorphisms. This section discusses infomorphisms. First, we give a concise mathematical definition, and then we discuss and formalize the various parts of this definition.
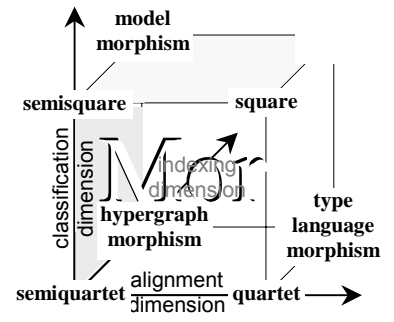
$$var(A_1) \underset{}{\overset{var(f)}{\rightleftarrows}} var(A_2) \qquad rel(A_1) \underset{}{\overset{rel(f)}{\rightleftarrows}} rel(A_2)$$

$$refer(A_1) \downarrow \quad refer(f) \quad \downarrow refer(A_2) \qquad sign(A_1) \downdownarrows \quad sign(f) \quad \downdownarrows sign(A_2)$$

$$ent(A_1) \underset{ent(f)}{\rightleftarrows} ent(A_2) \qquad sign(refer(A_1)) \underset{sign(refer(f))}{\rightleftarrows} sign(refer(A_2))$$

Figure 15: Model Morphism

○ An *infomorphism* (*morphism* of model-theoretic *structures*) $f = \langle refer(f), sign(f) \rangle : A_1 \rightleftarrows A_2$ from model A_1 to model A_2 (Figure 15) is a two dimensional construction consisting of a reference classification semisquare *refer(f)* and a signature classification square *sign(f)*, where the signature infomorphism of the reference semisquare is the vertical target of the signature square

$$sign(refer(f)) = vert\text{-}tgt(sign(f)).$$

```
(1) (SET$class infomorphism)
    (SET$class structure-morphism)
    (= structure-morphism infomorphism)

(2) (SET.FTN$function source)
    (= (SET.FTN$source source) infomorphism)
    (= (SET.FTN$target source) mod$model)

(3) (SET.FTN$function target)
    (= (SET.FTN$source target) infomorphism)
    (= (SET.FTN$target target) mod$model)

(4) (SET.FTN$function reference)
    (= (SET.FTN$source reference) infomorphism)
    (= (SET.FTN$target reference) cls.ssqr$semisquare)
    (= (SET.FTN$composition [reference cls.ssqr$horizontal-source])
       (SET.FTN$composition [source mod$reference]))
    (= (SET.FTN$composition [reference cls.sqr$horizontal-target])
       (SET.FTN$composition [target mod$reference]))

(5) (SET.FTN$function signature)
    (= (SET.FTN$source signature) infomorphism)
    (= (SET.FTN$target signature) cls.sqr$square)
    (= (SET.FTN$composition [signature cls.sqr$horizontal-source])
       (SET.FTN$composition [source mod$signature]))
    (= (SET.FTN$composition [signature cls.sqr$horizontal-target])
       (SET.FTN$composition [target mod$signature]))

(6) (= (SET.FTN$composition [signature cls.sqr$vertical-target])
       (SET.FTN$composition [reference cls.ssqr$signature]))
```

○ For convenience of theoretical presentation, we introduce additional model terminology for the composition between the reference function and the arity function for classification squares. Associated with an infomorphism $f : A_1 \rightarrow A_2$ is a *reference-arity* classification square *refer-arity(f)*.

```
(7) (SET.FTN$function reference-arity)
    (= (SET.FTN$source reference-arity) infomorphism)
    (= (SET.FTN$target reference-arity) cls.sqr$square)
    (= reference-arity (SET.FTN$composition [reference cls.ssqr$arity]))
```
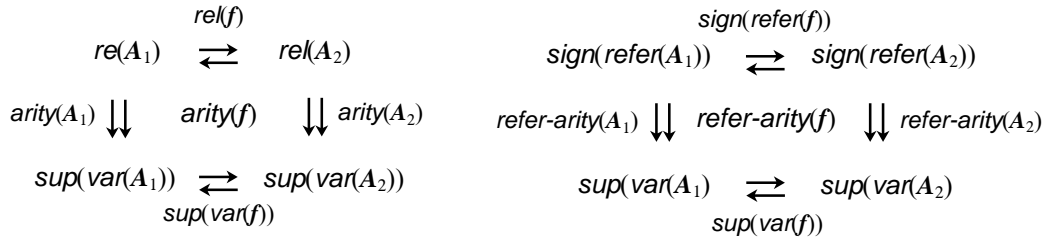
$$rel(f)$$
$$re(A_1) \rightleftarrows rel(A_2)$$

$$arity(A_1) \downdownarrows \quad arity(f) \quad \downdownarrows arity(A_2)$$

$$sup(var(A_1)) \rightleftarrows sup(var(A_2))$$
$$sup(var(f))$$

$$sign(refer(f))$$
$$sign(refer(A_1)) \rightleftarrows sign(refer(A_2))$$

$$refer\text{-}arity(A_1) \downdownarrows \quad refer\text{-}arity(f) \quad \downdownarrows refer\text{-}arity(A_2)$$

$$sup(var(A_1)) \rightleftarrows sup(var(A_2))$$
$$sup(var(f))$$

Diagram 8: Reference-arity and arity classification squares

○ The vertical composition (Diagram 8) of the signature classification square and the reference arity classification square defines an *arity* classification square $\#_f = arity(f) = sign(f) \cdot refer\text{-}arity(f)$.

```
(8) (SET.FTN$function arity)
    (= (SET.FTN$source arity) infomorphism)
    (= (SET.FTN$target arity) cls.sqr$square)
    (forall (?f (infomorphism ?f))
        (= (arity ?f)
           (cls.sqr$vertical-composition [(signature ?f) (reference-arity ?f)]))))
```

○ For convenience of reference, we introduce additional model theory terminology for the vertical source and target components of these classification squares. The vertical source of the signature classification square is called the *relation* infomorphism of f and denoted $rel(f)$. The infomorphism of the reference classification semisquare is called the *entity* infomorphism of f and denoted $ent(f)$. The invertible-pair of the reference classification semisquare is called the *variable* invertible pair of f and denoted $var(f)$.

In summary, an infomorphism has the following component infomorphisms:

–   the *entity* infomorphism $ent(f) = info(refer(f)) : ent(A_1) \rightleftarrows ent(A_2)$,

–   the *relation* infomorphism $rel(f) = vert\text{-}src(sign(f)) : rel(A_1) \rightleftarrows rel(A_2)$, and

–   the *variable* invertible pair $var(f) = inv\text{-}pr(refer(f)) : var(A_1) \rightleftarrows var(A_2)$.

Therefore, an infomorphism can be displayed as in Figure 1.

```
(9)  (SET.FTN$function entity)
     (= (SET.FTN$source entity) infomorphism)
     (= (SET.FTN$target entity) cls.info$infomorphism)
     (= entity (SET.FTN$composition [reference cls.ssqr$infomorphism]))

(10) (SET.FTN$function relation)
     (= (SET.FTN$source relation) infomorphism)
     (= (SET.FTN$target relation) cls.info$infomorphism)
     (= relation (SET.FTN$composition [signature cls.sqr$vertical-source]))

(11) (SET.FTN$function variable)
     (= (SET.FTN$source variable) infomorphism)
     (= (SET.FTN$target variable) set.invpr$invertible-pair)
     (= variable (SET.FTN$composition [reference cls.ssqr$invertible-pair]))
```

○ Following closely the definitions for models, let us define some further common terminology for any infomorphism $f : A_1 \rightarrow A_2$.

–   The *universe* function

$$univ(f) = inst(ent(f)) : univ(A_2) \rightarrow univ(A_1)$$

maps objects in the universe of A_2 to objects in the universe of A_1. The *entity* infomorphism (Figure 16) looks like

$$ent(f) = \langle univ(f), typ(ent(f)) \rangle : ent(A_1) \rightleftarrows ent(A_2),$$

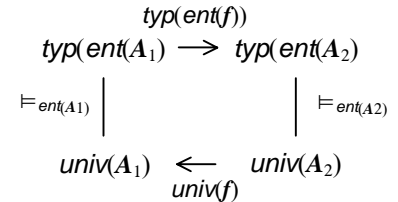and the universe and entity type functions satisfy the fundamental property of infomorphisms:

$$typ(ent(f))$$
$$typ(ent(A_1)) \longrightarrow typ(ent(A_2))$$
$$\vDash_{ent(A_1)} \qquad \qquad \vDash_{ent(A_2)}$$
$$univ(A_1) \longleftarrow univ(A_2)$$
$$univ(f)$$

Figure 16: Entity Infomorphism

$univ(f)(a_2) \vDash_{ent(A1)} \alpha_1$ iff $a_2 \vDash_{ent(A2)} typ(ent(f))(\alpha_1)$

for all objects $a_2 \in univ(A_2)$ and all entity types $\alpha_1 \in typ(ent(A_1))$.

– The *tuple* function

$tuple(f) = inst(rel(f)) : tuple(A_2) \to tuple(A_1)$

maps the relational tuples of A_2 to the relational tuples of A_1.
The *relation* infomorphism (Figure 17) looks like

$rel(f) = \langle tuple(f), typ(rel(f)) \rangle : rel(A_1) \rightleftarrows rel(A_2),$

and the tuple and relation type functions satisfy the fundamental property of infomorphisms:

$$typ(rel(f))$$
$$typ(rel(A_1)) \longrightarrow typ(rel(A_2))$$
$$\vDash_{rel(A1)} \quad\quad\quad \vDash_{rel(A2)}$$
$$tuple(A_1) \longleftarrow tuple(A_2)$$
$$tuple(f)$$

Figure 17: Relation Infomorphism

$tuple(f)(r_2) \vDash_{rel(A1)} \rho_1$ iff $r_2 \vDash_{rel(A2)} typ(rel(f))(\rho_1)$

for all tuples $r_2 \in tuple(A_2)$ and all relation types $\rho_1 \in typ(rel(A_1))$. Since the arities of both relation types and tuples are mapped by the power infomorphism of the invertible pair of f, valences of both relation types and tuple are preserved and arities are in bijective correspondence with their images.

```
(12) (SET.FTN$function universe)
     (= (SET.FTN$source universe) infomorphism)
     (= (SET.FTN$target universe) set.ftn$function)
     (= (SET.FTN$composition [universe set.ftn$source])
        (SET.FTN$composition [target mod$universe]))
     (= (SET.FTN$composition [universe set.ftn$target])
        (SET.FTN$composition [source mod$universe]))
     (= universe (SET.FTN$composition [entity cls.info$instance]))

(13) (SET.FTN$function tuple)
     (= (SET.FTN$source tuple) infomorphism)
     (= (SET.FTN$target tuple) set.ftn$function)
     (= (SET.FTN$composition [tuple set.ftn$source])
        (SET.FTN$composition [target mod$tuple]))
     (= (SET.FTN$composition [tuple set.ftn$target])
        (SET.FTN$composition [source mod$tuple]))
     (= tuple (SET.FTN$composition [relation cls.info$instance]))
```

○ Two infomorphisms are *composable* when the target of the first is equal to the source of the second. The *composition* of two composable infomorphisms $f_1 : A \to A'$ and $f_2 : A' \to A''$ is defined in terms of composition of their reference classification semisquares and the horizontal composition of their signature classification squares.

```
(14) (SET.LIM.PBK$opspan composable-opspan)
     (= (class1 composable-opspan) infomorphism)
     (= (class2 composable-opspan) infomorphism)
     (= (opvertex composable-opspan) mod$model)
     (= (first composable-opspan) target)
     (= (second composable-opspan) source)

(15) (REL$relation composable)
     (= (REL$class1 composable) infomorphism)
     (= (REL$class2 composable) infomorphism)
     (= (REL$extent composable) (SET.LIM.PBK$pullback composable-opspan))

(16) (SET.FTN$function composition)
     (= (SET.FTN$source composition) (SET.LIM.PBK$pullback composable-opspan))
     (= (SET.FTN$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))
              (= (reference (composition [?f1 ?f2]))
                 (cls.ssqr$composition [(reference ?f1) (reference ?f2)]))
              (= (signature (composition [?f1 ?f2]))
                 (cls.sqr$horizontal-composition [(signature ?f1) (signature ?f2)])))))
```

○ Composition satisfies the usual *associative law*.

The IFF Model Theory Ontology

```
(forall (?f1 (infomorphism ?f1)
         ?f2 (infomorphism ?f2)
         ?f3 (infomorphism ?f3)
         (composable ?f1 ?f2) (composable ?f2 ?f3))
    (= (composition [?f1 (composition [?f2 ?f3])])
       (composition [(composition [?f1 ?f2]) ?f3])))
```

o For any model *A*, there is an *identity* infomorphism.

```
(17) (SET.FTN$function identity)
     (= (SET.FTN$source identity) mod$model)
     (= (SET.FTN$target identity) infomorphism)
     (forall (?a (mod$model ?a))
         (and (= (source (identity ?a)) ?a)
              (= (target (identity ?a)) ?a)
              (= (reference (identity ?a))
                 (cls.ssqr$identity (mod$reference ?a)))
              (= (signature (identity ?a))
                 (cls.sqr$horizontal-identity (mod$signature ?a)))))
```

o The identity satisfies the usual *identity laws* with respect to composition.

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

○ A *simple* infomorphism is an infomorphism that has an identity variable invertible pair and an identity entity infomorphism.

```
(18) (SET$class simple)
     (SET$subclass simple infomorphism)
     (forall (?f (infomorphism ?f))
         (<=> (simple ?f)
             (and (= (mod$variable (source ?f)) (mod$variable (target ?f)))
                  (= (variable ?f) (set.invpr$identity (mod$variable (source ?f))))
                  (= (mod$entity (source ?f)) (mod$entity (target ?f)))
                  (= (entity ?f) (cls.info$identity (mod$entity (source ?f)))))))
```
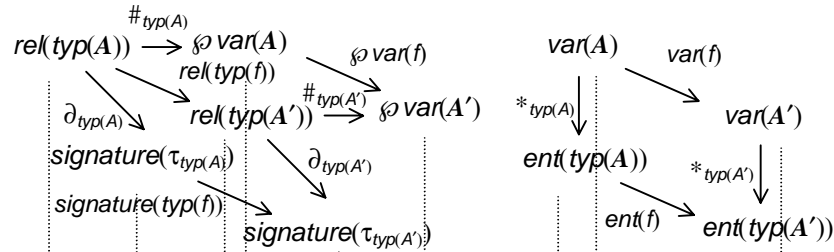
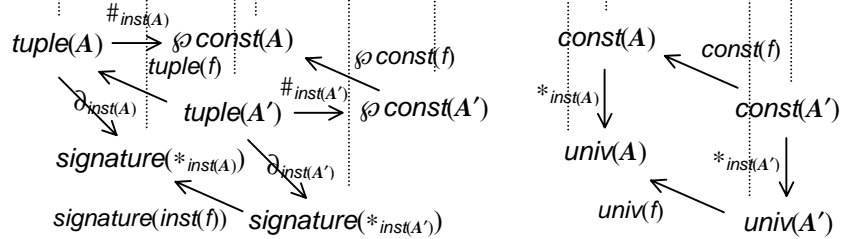

Figure 18: Type Language Morphism

Figure 19: Instance Hypergraph Morphism

○ An infomorphism $f = \langle refer(f), sign(f) \rangle : A \rightleftarrows A'$ defines two hypergraph morphisms,

– an *instance* hypergraph morphism $inst(f) : inst(A') \rightarrow inst(A)$ (Figure 18) and

– a *type* special language morphism $typ(f) : typ(A) \rightarrow typ(A')$ (Figure 19).

The IFF Model Theory Ontology

Of course, the special type language morphism is also a hypergraph morphism. The components of the hypergraph morphisms are defined in Table 5. Later, we will want to extend the special type language morphism to a special type language interpretation. The instance hypergraph morphism consists of

1. the universe function $univ(f) = node(inst(f)) : univ(A') \to univ(A)$, and
2. the tuple function $tuple(f) = rel(inst(f)) : tuple(A') \to tuple(A)$.

Table 5: Hypergraph Morphism Component Definitions

Hypergraph Morphism Component		Infomorphism Component	Notion
$refer(inst(f))$	=	$inst(refer(f))$	*semiquartet*
$sign(inst(f))$	=	$inst(sign(f))$	*quartet*
$arity(inst(f))$	=	$inst(arity(f))$	*quartet*
$name(inst(f))$	=	$inv(var(f))$	*bijection*
$node(inst(f))$	=	$univ(f)$	*function*
$edge(inst(f))$	=	$tuple(f)$	*function*
$refer(typ(f))$	=	$typ(refer(f))$	*quartet*
$sign(typ(f))$	=	$typ(sign(f))$	*quartet*
$arity(typ(f))$	=	$typ(arity(f))$	*quartet*
$var(typ(f))$	=	$dir(var(f))$	*bijection*
$ent(typ(f))$	=	$typ(ent(f))$	*function*
$rel(typ(f))$	=	$typ(rel(f))$	*function*

```
(19) (SET.FTN$function instance)
     (= (SET.FTN$source instance) infomorphism)
     (= (SET.FTN$target instance) hgph.mor$morphism)
     (= (SET.FTN$composition [instance hgph.mor$source])
        (SET.FTN$composition [source mod$instance]))
     (= (SET.FTN$composition [instance hgph.mor$target])
        (SET.FTN$composition [target mod$instance]))
     (= (SET.FTN$composition [instance hgph.mor$reference])
        (SET.FTN$composition [reference cls.ssqr$instance]))
     (= (SET.FTN$composition [instance hgph.mor$signature
        (SET.FTN$composition [signature cls.sqr$instance]))
     (= (SET.FTN$composition [instance hgph.mor$arity])
        (SET.FTN$composition [arity cls.sqr$instance]))
     (= (SET.FTN$composition [instance hgph.mor$name])
        (SET.FTN$composition [variable set.invpr$inverse]))
     (= (SET.FTN$composition [instance hgph.mor$node]) universe)
     (= (SET.FTN$composition [instance hgph.mor$edge]) tuple)

(20) (SET.FTN$function type)
     (= (SET.FTN$source type) infomorphism)
     (= (SET.FTN$target type) lang.mor.spl$special)
     (= (SET.FTN$composition [type lang.mor.spl$source])
        (SET.FTN$composition [source mod$type]))
     (= (SET.FTN$composition [type lang.mor.spl$target])
        (SET.FTN$composition [target mod$type]))
     (= (SET.FTN$composition [type lang.mor.spl$reference])
        (SET.FTN$composition [reference cls.ssqr$type]))
     (= (SET.FTN$composition [type lang.mor.spl$signature
        (SET.FTN$composition [signature cls.sqr$type]))
     (= (SET.FTN$composition [type lang.mor.spl$arity])
        (SET.FTN$composition [arity cls.sqr$type]))
     (= (SET.FTN$composition [type lang.mor.spl$variable])
        (SET.FTN$composition [variable set.invpr$direct]))
     (= (SET.FTN$composition [type lang.mor.spl$entity])
        (SET.FTN$composition [entity cls.info$type]))
     (= (SET.FTN$composition [type lang.mor.spl$relation])
        (SET.FTN$composition [relation cls.info$type]))
```

○ An infomorphism $f : A \rightleftharpoons A'$ defines an associated expression infomorphism

$$expr\text{-}info(f) = \langle tuple(f),\ ftn(typ(f))\rangle : expr\text{-}cls(A) \rightleftharpoons expr\text{-}cls(A')$$

between the expression classifications of its source and target models.

```
(21) (SET.FTN$function expression-infomorphism)
     (= (SET.FTN$source expression-infomorphism) infomorphism)
     (= (SET.FTN$target expression-infomorphism) cls.info$infomorphism)
     (= (SET.FTN$composition [expression-infomorphism cls.info$source])
        (SET.FTN$composition [source mod$expression-classification]))
     (= (SET.FTN$composition [expression-infomorphism cls.info$target])
        (SET.FTN$composition [target mod$expression-classification]))
     (= (SET.FTN$composition [expression-infomorphism cls.info$instance]) tuple)
     (= (SET.FTN$composition [expression-infomorphism cls.info$type])
        (SET.FTN$composition [type lang.expr.mor$function]))
```

○ Any infomorphism $f : A \rightleftharpoons A'$ can be extended to an *expression* infomorphism

$$expr(f) = \langle tuple(f),\ ftn(typ(f))\rangle : expr(A) \rightleftharpoons expr(A')$$

between the expression models of its source and target, whose reference semisquare is the reference semisquare of f, but whose relation infomorphism is the expression infomorphism for f. The type language morphism of the expression infomorphism is the expression language morphism of the type language morphism: the model expression function commutes with the underlying type function and the language expression function. The expression operator commutes with source and target embeddings.

$$ftn(typ(f))$$
$$set(typ(L_1)) \longrightarrow set(typ(L_2))$$
$$\Big\vDash expr\text{-}cls(A1) \qquad \Big\vDash expr\text{-}cls(A2)$$
$$tuple(L_1) \longleftarrow tuple(L_2)$$
$$tuple(f)$$

Figure 20: Expression Infomorphism

```
(22) (SET.FTN$function expression)
     (= (SET.FTN$source expression) infomorphism)
     (= (SET.FTN$target expression) infomorphism)
     (= (SET.FTN$composition [expression source])
        (SET.FTN$composition [source mod$expression]))
     (= (SET.FTN$composition [expression target])
        (SET.FTN$composition [target mod$expression]))
     (= (SET.FTN$composition [expression reference]) reference)
     (= (SET.FTN$composition [(SET.FTN$composition
            [expression signature]) cls.sqr$vertical-source])
          expression-infomorphism)
     (= (SET.FTN$composition [(SET.FTN$composition
            [expression signature]) cls.sqr$vertical-target])
        (SET.FTN$composition [signature cls.sqr$vertical-target]))

     (= (SET.FTN$composition [expression type])
        (SET.FTN$composition [type lang.expr.mor$expression]))

     (forall (?f (infomorphism ?f))
         (= (mod.info$composition [(mod$embedding (source ?f)) (expression ?f)])
            (mod.info$composition [?f (mod$embedding (target ?f))]))))
```

○ Any infomorphism $f : A_1 \rightleftharpoons A_2$ defines a function $thm(f) : thm(A_1) \to thm(A_2)$ between the theorem classes of source and target models. This is well-defined, since the (expression function of the) associated type language morphism $typ(f) : typ(A_1) \to typ(A_2)$ maps theorems of A_1 to theorems of A_2: $ftn(typ(f))(thm(A_1)) \subseteq thm(A_2)$. Figure 20 depicts the expression infomorphism, and the following expresses the fundamental condition for this infomorphism:

Let φ_1 be any theorem of A_1. Since $tuple(f)(r_2) \vDash \varphi_1$ iff $r_2 \vDash ftn(typ(f))(\varphi_1)$ for any tuple r_2 of A_2, clearly $ftn(typ(f))(\varphi_1)$ is a theorem of A_2.

```
(23) (KIF$function theorem-function)
     (= (KIF$source theorem-function) infomorphism)
     (= (KIF$target theorem-function) SET.FTN$function)
     (forall (?f (infomorphism ?f)
```

```
                    (and (= (SET.FTN$source (theorem-function ?f)) (mod$theorem (source ?f)))
                         (= (SET.FTN$target (theorem-function ?f)) (mod$theorem (target ?f)))
                         (SET.TN$restriction
                             (theorem-function ?f)
                             (set.ftn$power (lang.expr.mor$function (type ?f))))))))
```

○   Any infomorphism $f = \langle refer(f), sign(f) \rangle : A_1 \rightleftarrows A_2$ defines a function $th(f) : th(A_1) \to th(A_2)$ between the theory classes of source and target models.

```
(24) (KIF$function theory-function)
     (= (KIF$source theory-function) infomorphism)
     (= (KIF$target theory-function) SET.FTN$function)
     (forall (?f (infomorphism ?f)
        (and (= (SET.FTN$source (theory-function ?f)) (mod$theory (source ?f)))
             (= (SET.FTN$target (theory-function ?f)) (mod$theory (target ?f)))
             (forall (?t ((mod$theory (source ?f)) ?t))
                (= (th$axiom ((theory-function ?f) ?t))
                   ((power (theorem-function ?f)) (axiom ?t))))))))
```

○   For any infomorphism $f = \langle refer(f), sign(f) \rangle : A_1 \rightleftarrows A_2$ there is a maximal special theory morphism $maxth(f) : maxth(A_1) \to maxth(A_2)$, whose underlying (base) special language morphism is the type language morphism of f: $base(maxth(f)) = typ(f)$. This is well defined, since by the comments above the theorems of A_1 are mapped to the theorems of A_2.

```
(25) (SET.FTN$function maximal-theory)
     (= (SET.FTN$source maximal-theory) infomorphism)
     (= (SET.FTN$target maximal-theory) th.mor.spl$special)
     (= (SET.FTN$composition [maximal-theory th.mor.spl$source])
        (SET.FTN$composition [source mod$maximal-theory]))
     (= (SET.FTN$composition [maximal-theory th.mor.spl$target])
        (SET.FTN$composition [target mod$maximal-theory]))
     (= (SET.FTN$composition [maximal-theory th.mor.spl$base]) type)
```

○   Every infomorphism $f : A_1 \to A_2$ generates a *logic infomorphism* $log(f) : log(A_1) \to log(A_2)$, whose component infomorphism is f and whose theory morphism $th(log(f)) = th(f) : th(A_1) \to th(A_2)$ is the maximal theory morphism generated by f.

```
(26) (SET.FTN$function logic)
     (= (SET.FTN$source logic) infomorphism)
     (= (SET.FTN$target logic) log.info$infomorphism)
     (= (SET.FTN$composition [logic log.info$model]) (SET.FTN$identity infomorphism))
     (= (SET.FTN$composition [logic log.info$theory]) maximal-theory)
```

○   For any infomorphism $f : A_1 \to A_2$, the *empty prologic infomorphism* has model component f and the empty theory morphism over $typ(f)$ as its theory component. This is actually a logic infomorphism.

```
(27) (SET.FTN$function empty)
     (= (SET.FTN$source empty) infomorphism)
     (= (SET.FTN$target empty) plog.info$infomorphism)
     (= (SET.FTN$composition [empty plog.info$source])
        (SET.FTN$composition [source mod$empty]))
     (= (SET.FTN$composition [empty plog.info$target])
        (SET.FTN$composition [target mod$empty]))
     (= (SET.FTN$composition [empty plog.info$model])
        (SET.FTN$identity infomorphism))
     (= (SET.FTN$composition [empty plog.info$theory])
        (SET.FTN$composition [type lang.mor$empty]))
```

○   For any infomorphism $f : A_1 \to A_2$, the *full prologic infomorphism* has model component f and the full theory morphism over $typ(f)$ as its theory component.

```
(28) (SET.FTN$function full)
     (= (SET.FTN$source full) infomorphism)
     (= (SET.FTN$target full) plog.info$infomorphism)
     (= (SET.FTN$composition [full plog.info$source])
        (SET.FTN$composition [source mod$full]))
     (= (SET.FTN$composition [full plog.info$target])
        (SET.FTN$composition [target mod$full]))
     (= (SET.FTN$composition [full plog.info$model])
```

```
        (SET.FTN$identity infomorphism))
(= (SET.FTN$composition [full plog.info$theory])
   (SET.FTN$composition [type lang.mor$full]))
```