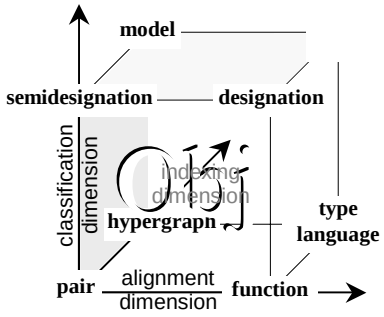


The Namespace of Models

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	
		case indication projection comediator spanmodel	
mod .mor	model-morphism = structure-morphism simple	source target reference signature reference-arity arity entity relation variable universe tuple composition identity instance type	composable-opspan composable

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.

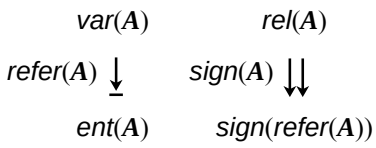


Figure 1: Model

- A *model* (model-theoretic *structure*) $\boldsymbol{A} = \langle refer(\boldsymbol{A}), sign(\boldsymbol{A})\rangle$ is a hypergraph of classifications – a two dimensional construction (Figure 1) consisting of
  - a *reference* semidesignation $\tau_{\boldsymbol{A}} = refer(\boldsymbol{A}) = \langle inst(refer(\boldsymbol{A})), typ(refer(\boldsymbol{A}))\rangle : var(\boldsymbol{A}) \nrightarrow ent(\boldsymbol{A})$ and
  - a *signature* designation $\partial_{\boldsymbol{A}} = sign(\boldsymbol{A}) = \langle inst(sign(\boldsymbol{A})), typ(sign(\boldsymbol{A}))\rangle : rel(\boldsymbol{A}) \rightrightarrows sign(refer(\boldsymbol{A}))$,

  where the signature classification of the reference semidesignation is the target classification of the signature designation

  $sign(refer(\boldsymbol{A})) = (tgt(sign(\boldsymbol{A}))$.

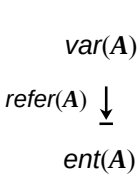


Figure 2a: Reference Semidesignation – abstract

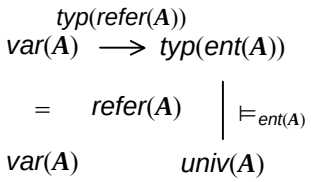


Figure 2b: Reference Semidesignation – details

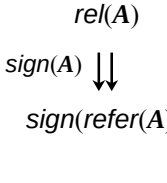


Figure 3a: Signature Designation - abstract

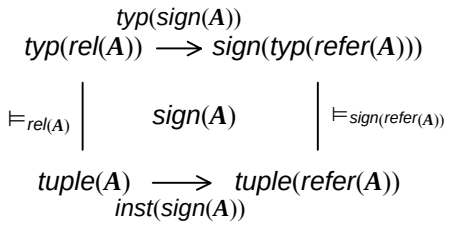


Figure 3b: Signature Designation - details

```
(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]))
```

○

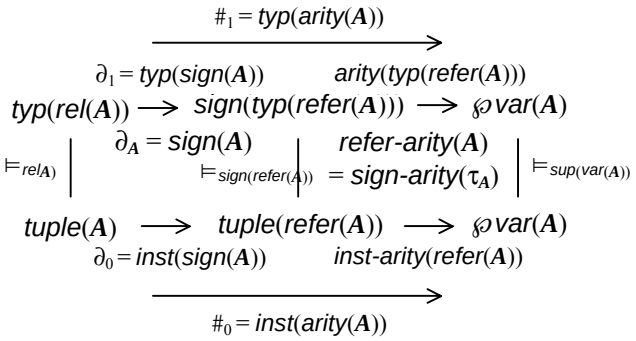


Diagram 1: Signature, reference-arity and arity designations

○ 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-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 $\#_{\boldsymbol{A}} = \langle \#_0, \#_1 \rangle =$ ***arity***(***A***) = ***sign***(***A***) · ***refer-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 $\partial_{\boldsymbol{A}} = \langle \partial_0, \partial_1 \rangle =$ ***sign***(***A***) : ***rel***(***A***) → ***sign***(***refer***(***A***)),

– *arity* designation $\#_{\boldsymbol{A}} = \langle \#_0, \#_1 \rangle =$ ***arity***(***A***) : ***rel***(***A***) → ***sup***(***var***(***A***)), and

– *reference* semidesignation $\tau_{\boldsymbol{A}} =$ ***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***))×***Natno***. Then the reference type function is the projection function

typ(***refer***(***A***)) = π_1 : ***typ***(***ent***(***A***))×***Natno*** → ***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 $\mathit{univ}(\boldsymbol{A}) = \mathit{inst}(\mathit{ent}(\boldsymbol{A}))$ of entity instances is called the *universe* (or *universal domain*) for $\boldsymbol{A}$. In the entity classification $\mathit{ent}(\boldsymbol{A})$, the elements of the universe are classified by entity types. Elements of the set of relational instances $\mathit{tuple}(\boldsymbol{A}) = \mathit{inst}(\mathit{rel}(\boldsymbol{A}))$ are called *tuples* or (*relational*) *arguments*. In the relation classification $\mathit{rel}(\boldsymbol{A})$, tuples are classified by relation types. Although we do not specify it, usually there is a subset of individuals $\mathit{indiv}(\boldsymbol{A}) \subseteq \mathit{univ}(\boldsymbol{A})$, 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 $r \in \mathit{tuple}(\boldsymbol{A})$ are abstract, and by themselves are typeless. They are associated with concrete reference tuples, and thus elements of the universe, via the tuple function

  $\partial_0 = \mathit{inst}(\mathit{sign}(\boldsymbol{A})) : \mathit{tuple}(\boldsymbol{A}) \to \mathit{tuple}(\mathit{refer}(\boldsymbol{A}))$.

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

  $\#_0 = \mathit{inst}(\mathit{arity}(\boldsymbol{A})) : \mathit{tuple}(\boldsymbol{A}) \to \wp\, \mathit{var}(\boldsymbol{A})$.
- The entity classification $\mathit{ent}(\boldsymbol{A}) = \langle \mathit{univ}(\boldsymbol{A}), \mathit{typ}(\mathit{ent}(\boldsymbol{A})), \vDash_{\mathit{ent}(\boldsymbol{A})} \rangle$ has the entity types of $\boldsymbol{A}$ as its types and the elements of the universe of $\boldsymbol{A}$ as its instances. The relation classification $\mathit{rel}(\boldsymbol{A}) = \langle \mathit{tuple}(\boldsymbol{A}), \mathit{typ}(\mathit{rel}(\boldsymbol{A})), \vDash_{\mathit{rel}(\boldsymbol{A})} \rangle$ has the relation types of $\boldsymbol{A}$ as its types and the tuples of $\boldsymbol{A}$ 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, $r \vDash_{\mathit{rel}(\boldsymbol{A})} \rho$, then the concrete tuple $\partial_0(r)$ has the signature type $\partial_1(\rho)$, $\partial_0(r) \vDash_{\mathit{sign}(\mathit{ref}(\boldsymbol{A}))} \partial_1(\rho)$; this means that the arities are related as $\mathit{arity}(\rho) \subseteq \mathit{arity}(r)$, and $\partial_0(r)(j(x)) \vDash_{\mathit{ent}(\boldsymbol{A})} \partial_1(\rho)(x)$ for each variable $x \in \mathit{arity}(\rho)$. Of course, although a tuple r has a fixed arity $\mathit{arity}(r) = \{x_1, x_2, \ldots x_m\}$ and a fixed signature $\partial_0(r) = \{\partial_0(r)(x) \mid x \in \mathit{arity}(r)\} = (\partial_0(r)(x_1), \partial_0(r)(x_2), \ldots \partial_0(r)(x_m))$, a different relational classification incidence $r \vDash_{\mathit{rel}(\boldsymbol{A})} \rho'$ may assert $\mathit{arity}(\rho') \subseteq \mathit{arity}(r)$, and $\partial_0(r)(j'(x')) \vDash_{\mathit{ent}(\boldsymbol{A})} \partial_1(\rho')(x')$ for each variable $x' \in \mathit{arity}(\rho')$, where even the cardinalities (valences) differ $|\mathit{arity}(\rho)| \neq |\mathit{arity}(\rho')|$. 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]))
```

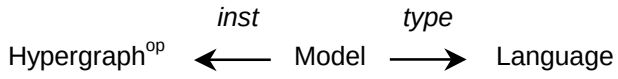


Diagram 2: The Hypergraph Span of a Model

- There is a derived view of a model as a "classification" of hypergraphs – the hypergraphs are classified componentwise. A model $\boldsymbol{A}$ with components

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

  defines two hypergraphs (Diagram 2), an *instance* hypergraph $inst(\boldsymbol{A})$ and a *type* language $typ(\boldsymbol{A})$. 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) $node(inst(\boldsymbol{A})) = inst(ent(\boldsymbol{A})) = universe(\boldsymbol{A})$,
  2. a set of edges (relation tuples) $edge(inst(\boldsymbol{A})) = inst(rel(\boldsymbol{A})) = tuple(\boldsymbol{A})$, and
  3. an *signature* function $\partial_0 : tuple(\boldsymbol{A}) \to tuple(refer(\boldsymbol{A}))$ that maps an edge $t \in tuple(\boldsymbol{A})$ to its vertex $\partial_0(t)$ of nodes.

Table 3: Hypergraph Component Definitions

Hypergraph Component		Classification Component	Notion
$refer(inst(\boldsymbol{A}))$	=	$inst(refer(\boldsymbol{A}))$	*set pair*
$sign(inst(\boldsymbol{A}))$	=	$inst(sign(\boldsymbol{A}))$	*function*
$arity(inst(\boldsymbol{A}))$	=	$inst(arity(\boldsymbol{A}))$	*function*
$name(inst(\boldsymbol{A}))$	=	$var(\boldsymbol{A})$	*set*
$node(inst(\boldsymbol{A}))$	=	$univ(\boldsymbol{A})$	*set*
$edge(inst(\boldsymbol{A}))$	=	$tuple(\boldsymbol{A})$	*set*
$refer(typ(\boldsymbol{A}))$	=	$typ(refer(\boldsymbol{A}))$	*function*
$sign(typ(\boldsymbol{A}))$	=	$typ(sign(\boldsymbol{A}))$	*function*
$arity(typ(\boldsymbol{A}))$	=	$typ(arity(\boldsymbol{A}))$	*function*
$var(typ(\boldsymbol{A}))$	=	$var(\boldsymbol{A})$	*set*
$ent(typ(\boldsymbol{A}))$	=	$typ(ent(\boldsymbol{A}))$	*set*
$rel(typ(\boldsymbol{A}))$	=	$typ(rel(\boldsymbol{A}))$	*set*

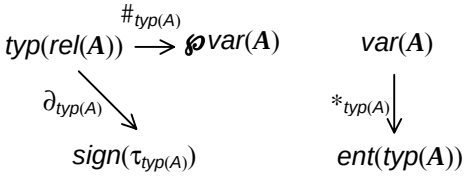


Figure 4: Type Language

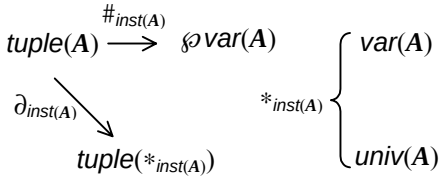


Figure 5: Instance Hypergraph

```
(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)
```

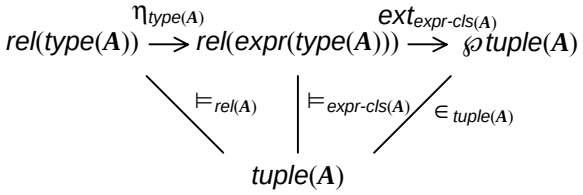


Diagram 3: Expression Classification & Extension Function

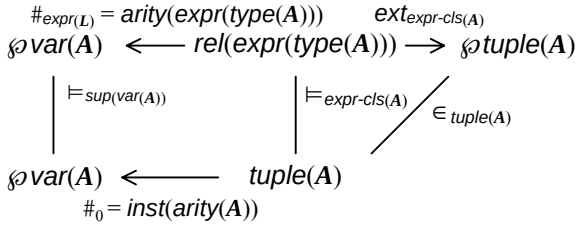


Diagram 4: Arity & Extension Functions

- For any model $\boldsymbol{A}$, we extend the relation classification

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

  to an *expression* classification

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

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

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

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

  There is a precursor function

  $cmpt(\boldsymbol{A}) : rel(expr(type(\boldsymbol{A}))) \to \wp tuple(\boldsymbol{A})$

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

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

  Any tuple classifiable by φ is also compatible with φ: $ext(expr\text{-}cls(\boldsymbol{A}))(\varphi) \subseteq cmpt(\boldsymbol{A})(\varphi)$. 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

  $\bullet cmpt(\boldsymbol{A}) : rel(expr(type(\boldsymbol{A}))) \to \wp tuple(\boldsymbol{A})$

  where $t \in \bullet cmpt(\boldsymbol{A})(\varphi)$ **means** $inst(arity(\boldsymbol{A}))(t) = arity(expr(\boldsymbol{L}))(\varphi)$.

```
(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 $\varphi \in rel(expr(type(\boldsymbol{A})))$ there is a surjective, idempotent trim function from the set of compatible tuples $cmpt(\boldsymbol{A})(\varphi)$ to the set of strictly compatible tuples $\bullet cmpt(\boldsymbol{A})(\varphi)$

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

  This function trims each tuple $t \in cmpt(\boldsymbol{A})(\varphi)$ returning only the part of t that is essential in the classification $t \vDash_{expr\text{-}cls(\boldsymbol{A})} \varphi$. Thus, $trim(\boldsymbol{A})(\varphi)$ 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 $\rho \in \mathit{rel}(\mathit{type}(\boldsymbol{A}))$ be any relation type (primitive expression) and let $t \in \mathit{tuple}(\boldsymbol{A})$ be any tuple.
  $t \in \mathit{ext}_{\mathit{expr\text{-}cls}(\boldsymbol{A})}(\rho)$ iff $t \in \mathit{ext}_{\mathit{rel}(\boldsymbol{A})}(\rho)$.

```
(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 $\varphi \in \mathit{rel}(\mathit{expr}(\mathit{type}(\boldsymbol{A})))$ be any expression and let $t \in \mathit{tuple}(\boldsymbol{A})$ be any tuple.
  $t \in \mathit{ext}_{\mathit{expr\text{-}cls}(\boldsymbol{A})}(\neg\varphi)$ iff $t \in \mathit{cmpt}(\boldsymbol{A})(\neg\varphi)$ and $t \notin \mathit{ext}_{\mathit{expr\text{-}cls}(\boldsymbol{A})}(\varphi)$.

```
(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

$\wedge$	$\mapsto$	$\cap_{\mathit{tuple}(\boldsymbol{A})} : \wp\mathit{tuple}(\boldsymbol{A}) \times \wp\mathit{tuple}(\boldsymbol{A}) \to \wp\mathit{tuple}(\boldsymbol{A})$
$\vee$	$\mapsto$	$\cup_{\mathit{tuple}(\boldsymbol{A})} : \wp\mathit{tuple}(\boldsymbol{A}) \times \wp\mathit{tuple}(\boldsymbol{A}) \to \wp\mathit{tuple}(\boldsymbol{A})$
$\Rightarrow$	$\mapsto$	$\Rightarrow_{\mathit{tuple}(\boldsymbol{A})} : \wp\mathit{tuple}(\boldsymbol{A}) \times \wp\mathit{tuple}(\boldsymbol{A}) \to \wp\mathit{tuple}(\boldsymbol{A})$
$\Leftrightarrow$	$\mapsto$	$\Leftrightarrow_{\mathit{tuple}(\boldsymbol{A})} : \wp\mathit{tuple}(\boldsymbol{A}) \times \wp\mathit{tuple}(\boldsymbol{A}) \to \wp\mathit{tuple}(\boldsymbol{A})$

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

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

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

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

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

```
(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)])])))))
```

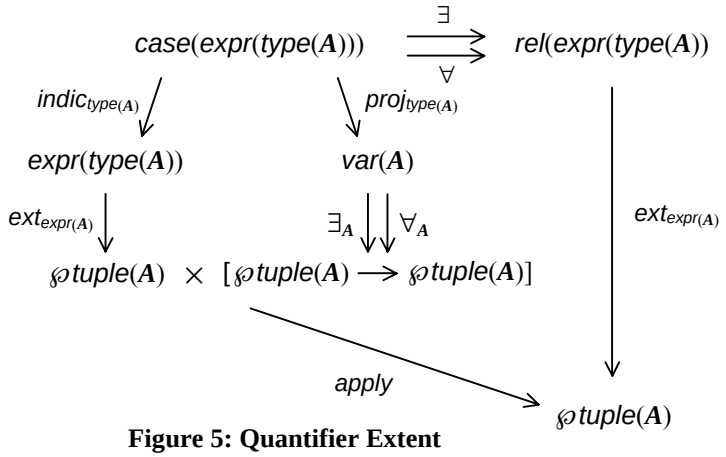


Figure 5: Quantifier Extent

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

$\pi_{\boldsymbol{A}}(x) : tuple(\boldsymbol{A}) \to tuple(\boldsymbol{A})$

and the existential and universal functions

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

where

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

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

for any subset of tuples $X \in \wp tuple(\boldsymbol{A})$.

[*existential quantification extent*]

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

$\exists s \in \mathit{tuple}(\boldsymbol{A})$ such that $\pi_{\boldsymbol{A}}(x)(s) = t$ and $s \in \mathit{ext}_{\mathit{expr\text{-}cls}(\boldsymbol{A})}(\varphi)$.

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

[*universal quantification extent*]

$t \in \mathit{ext}_{\mathit{expr\text{-}cls}(\boldsymbol{A})}((\forall x)\varphi)$ iff

$\forall s \in \mathit{tuple}(\boldsymbol{A})$ if $\pi_{\boldsymbol{A}}(x)(s) = t$ then $s \in \mathit{ext}_{\mathit{expr\text{-}cls}(\boldsymbol{A})}(\varphi)$.

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

```
(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 (typle ?a)) (set$power (typle ?a))])]))))))
```

– [*substitution extent*]

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

$\mathit{dom}(\mathit{type}(\boldsymbol{A}))(h) = \mathit{arity}(\mathit{expr}(\mathit{type}(\boldsymbol{A})))(\varphi)$.

There is a surjective, idempotent trim operation

$\mathit{trim}(\boldsymbol{A})(\varphi) : \mathit{cmpt}(\boldsymbol{A})(\varphi) \to \bullet\mathit{cmpt}(\boldsymbol{A})(\varphi)$

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

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

The strict tuple substitution function

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

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

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

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

The full extent is then defined as:

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

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

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

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

Then

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

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

```
(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, the *extent* function is defined as the cotupling of an *extent tuple* of functions. And the expression classification 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 $\boldsymbol{A} = \langle \mathit{refer}(\boldsymbol{A}), \mathit{sign}(\boldsymbol{A}) \rangle$ can be extended to an *expression* model $\mathit{expr}(\boldsymbol{A})$, whose reference semidesignation is the same, but whose relation classification is the expression classification for $\boldsymbol{A}$.
  - $\mathit{refer}(\mathit{expr}(\boldsymbol{A})) = \mathit{refer}(\boldsymbol{A})$
    $= \langle \mathit{inst}(\mathit{refer}(\boldsymbol{A})), \mathit{typ}(\mathit{refer}(\boldsymbol{A})) \rangle : \mathit{var}(\boldsymbol{A}) \mapsto \mathit{ent}(\boldsymbol{A})$,
  - $\mathit{sign}(\mathit{expr}(\boldsymbol{A}))$
    $= \langle \mathit{inst}(\mathit{sign}(\boldsymbol{A})), \mathit{sign}(\mathit{expr}(\mathit{type}(\boldsymbol{A}))) \rangle : \mathit{expr\text{-}cls}(\boldsymbol{A}) \rightleftarrows \mathit{sign}(\mathit{refer}(\boldsymbol{A}))$.

  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 1).

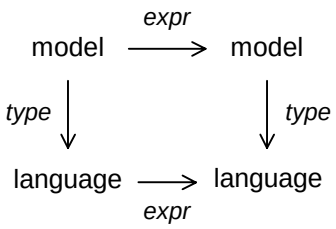


Diagram 6: Commutation of expression with type

```
(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* model morphism from any model $\boldsymbol{A}$ to its expression model $\mathit{expr}(\boldsymbol{A})$, whose relation infomorphism has identity instance function and primitive embedding type function.

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

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

```
(33) (SET.FTN$function embedding)
     (= (SET.FTN$source expression) model)
     (= (SET.FTN$target expression) mod.mor$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 $\boldsymbol{L}$ and sentences of that same language. An $\boldsymbol{L}$-model $\boldsymbol{A} \in \mathit{mod}(\boldsymbol{L})$ *satisfies* an $\boldsymbol{L}$-sentence $\sigma \in \mathit{sent}(\boldsymbol{L})$,

$$\boldsymbol{A} \models_{\boldsymbol{L}} \sigma,$$

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

σ *holds in* $\boldsymbol{A}$
$\boldsymbol{A}$ *satisfies* σ
σ *is satisfied in* $\boldsymbol{A}$
$\boldsymbol{A}$ *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 $\boldsymbol{A}$ since they have been defined for the relation types of $\mathit{expr}(\boldsymbol{A})$. 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 $\rho \in \mathit{rel}(\mathit{typ}(\boldsymbol{A}))$. The type signature function

$$\partial_1 : \mathit{typ}(\mathit{rel}(\boldsymbol{A})) \to \mathit{sign}(\mathit{typ}(\mathit{refer}(\boldsymbol{A})))$$

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

$$\partial_0 : \mathit{inst}(\mathit{rel}(\boldsymbol{A})) \to \mathit{tuple}(\mathit{refer}(\boldsymbol{A}))$$

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

$$\mathit{ext}_{\mathit{rel}(\boldsymbol{A})}(\rho) \subseteq \partial_0^{-1}(\mathit{ext}_{\mathit{sign}(\mathit{refer}(\boldsymbol{A}))}(\partial_1(\rho))).$$

We say that $\boldsymbol{A}$ *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

$$\partial_{\boldsymbol{A}} = \langle \partial_0, \partial_1 \rangle = \mathit{sign}(\boldsymbol{A}) : \mathit{rel}(\boldsymbol{A}) \rightrightarrows \mathit{sign}(\mathit{refer}(\boldsymbol{A})).$$

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 $\rho \in \mathit{rel}(\mathit{typ}(\boldsymbol{A}))$, a subset of tuples, may not correspond exactly to the extent of the signature $\partial_1(\rho) \in \mathit{sign}(\mathit{typ}(\mathit{refer}(\boldsymbol{A})))$. 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:

  $r \models_{\mathit{rel}(\boldsymbol{A})} \rho$ implies $\partial_0(r) \models_{\mathit{sign}(\mathit{refer}(\boldsymbol{A}))} \partial_1(\rho)$

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

So, a classification incidence $\partial_0(r) \models_{sign(refer(\boldsymbol{A}))} \partial_1(\rho)$ 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

$r \not\models_{rel(\boldsymbol{A})} \rho$.

Representation can overcome this constraint.

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

$r \models_{rel(\boldsymbol{A})} \rho$.

Satisfaction can overcome this constraint.

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

1. There is an abstract tuple $r \in tuple(\boldsymbol{A})$ such that $r \models_{rel(A)} \rho$ and $a = \partial_0(r)$.
2. There is an abstract tuple $r \in tuple(\boldsymbol{A})$ such that $a = \partial_0(r)$, but $r \not\models_{rel(\boldsymbol{A})} \rho$.
3. There is no abstract tuple $r \in tuple(\boldsymbol{A})$ with $r \models_{rel(A)} \rho$ and $a \leq \partial_0(r)$.

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

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

– We say that a structure $\boldsymbol{A}$ *strongly satisfies* a relation type $\rho \in rel(typ(\boldsymbol{A}))$ 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)])))
```

Case and Spanmodels

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

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

```
(41) (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 $\boldsymbol{A} = \langle \mathit{refer}(\boldsymbol{A}), \mathit{sign}(\boldsymbol{A}) \rangle$ has an associated *indication* designation

  $\mathit{indic}(\boldsymbol{A})$

  $= \langle \mathit{indic}(\mathit{inst}(\boldsymbol{A})), \mathit{indic}(\mathit{typ}(\boldsymbol{A})) \rangle : \mathit{case}(\boldsymbol{A}) \rightrightarrows \mathit{rel}(\boldsymbol{A})$,

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

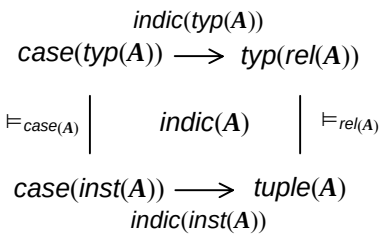


Figure 6: Indication designation

```
(42) (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 $\boldsymbol{A} = \langle \mathit{refer}(\boldsymbol{A}), \mathit{sign}(\boldsymbol{A}) \rangle$ has an associated *projection* designation

  $\mathit{proj}(\boldsymbol{A})$

  $= \langle \mathit{proj}(\mathit{inst}(\boldsymbol{A})), \mathit{proj}(\mathit{typ}(\boldsymbol{A})) \rangle : \mathit{case}(\boldsymbol{A}) \rightrightarrows \mathit{cla}(\mathit{var}(\boldsymbol{A}))$,

  whose source is the case classification of $\boldsymbol{A}$, whose target is the identity classification of the variable set of $\boldsymbol{A}$, 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.

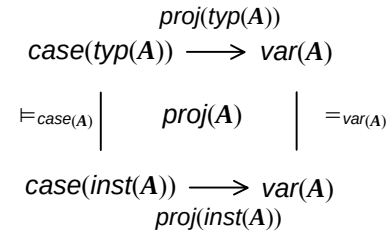


Figure 7: Projection designation

```
(43) (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 $\boldsymbol{A} = \langle refer(\boldsymbol{A}), sign(\boldsymbol{A}) \rangle$ has an associated *comediator* designation

  $*_{\boldsymbol{A}} = comed(\boldsymbol{A})$

  $= \langle comed(inst(\boldsymbol{A})), comed(typ(\boldsymbol{A})) \rangle : case(\boldsymbol{A}) \rightrightarrows ent(\boldsymbol{A})$,

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

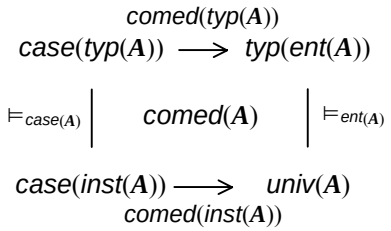


Figure 8: Comediator designation

```
(44) (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]))
```

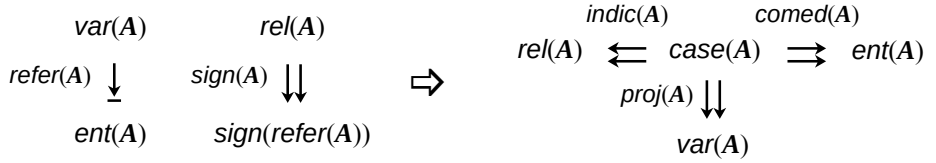


Diagram 7a: Model to Spanmodel - abstract version

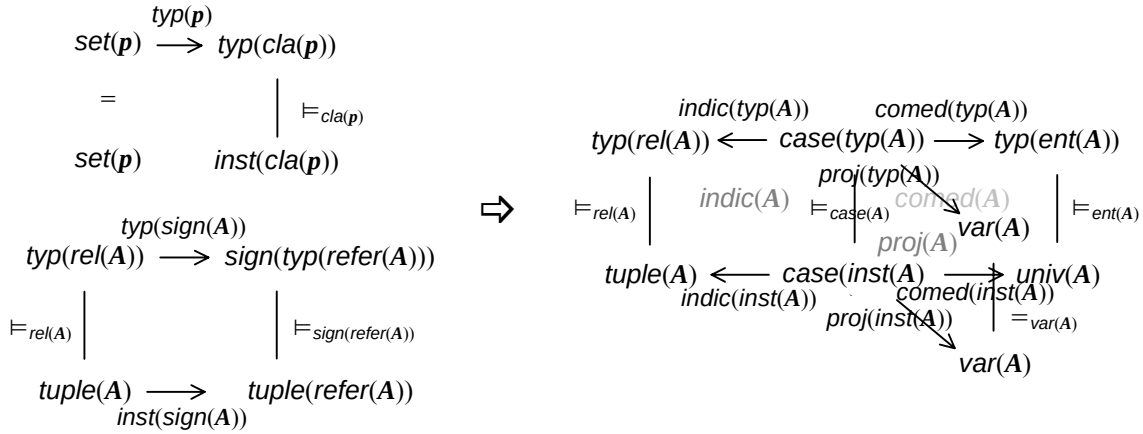
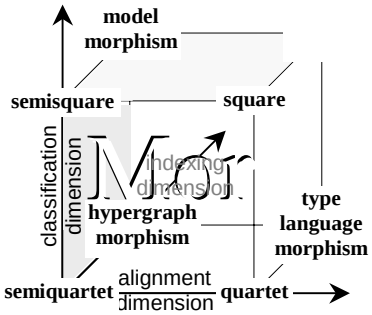


Diagram 7b: Model to Spanmodel - detailed version

- Associated with any model $\boldsymbol{A} = \langle refer(\boldsymbol{A}), sign(\boldsymbol{A}) \rangle$ is a spanmodel $spnmod(\boldsymbol{A})$, whose vertex (prehension) classification is the case (role) classification of $\boldsymbol{A}$,
  - whose first designation (actuality) is the comediator designation $1^{st}_{spnmod(\boldsymbol{A})} = comed(\boldsymbol{A})$ with target classification being the entity classification of $\boldsymbol{A}$,
  - whose second designation (naming) is the projection designation $2^{nd}_{spnmod(\boldsymbol{A})} = proj(\boldsymbol{A})$ with target classification being the variable identity classification of $\boldsymbol{A}$, and
  - whose third designation (nexus) is the indication designation $3^{rd}_{spnmod(\boldsymbol{A})} = indic(\boldsymbol{A})$ with target classification being the relation classification of $\boldsymbol{A}$.

```
(45) (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))))
```



Model Morphisms

`mod.mor`

Models are connected by and comparable with model morphisms. This section discusses model morphisms. First, we give a concise mathematical definition, and then we discuss and formalize the various parts of this definition.

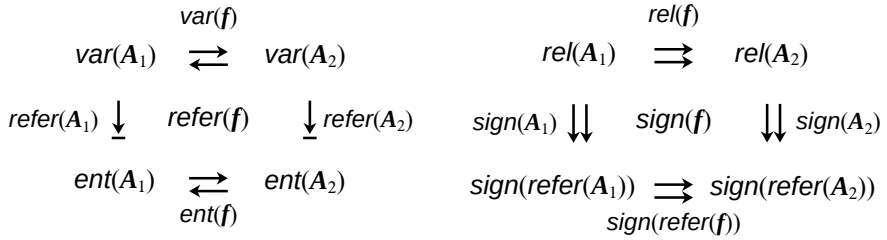


Figure 9: Model Morphism

- A *model morphism* (*morphism* of model-theoretic *structures*) $\boldsymbol{f} = \langle \mathit{refer}(\boldsymbol{f}), \mathit{sign}(\boldsymbol{f}) \rangle : \boldsymbol{A}_1 \to \boldsymbol{A}_2$ from model $\boldsymbol{A}_1$ to model $\boldsymbol{A}_2$ (Figure 9) is a two dimensional construction consisting of a reference classification semisquare $\mathit{refer}(\boldsymbol{f})$ and a signature classification square $\mathit{sign}(\boldsymbol{f})$, where the signature infomorphism of the reference semisquare is the vertical target of the signature square

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

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

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

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

(4) (SET.FTN$function reference)
    (= (SET.FTN$source reference) model-morphism)
    (= (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) model-morphism)
    (= (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 a model morphism $\boldsymbol{f} : \boldsymbol{A}_1 \to \boldsymbol{A}_2$ is a *reference-arity* classification square $\mathit{refer\text{-}arity}(\boldsymbol{f})$.

```
(7) (SET.FTN$function reference-arity)
    (= (SET.FTN$source reference-arity) model-morphism)
    (= (SET.FTN$target reference-arity) cls.sqr$square)
    (= reference-arity (SET.FTN$composition [reference cls.ssqr$arity]))
```

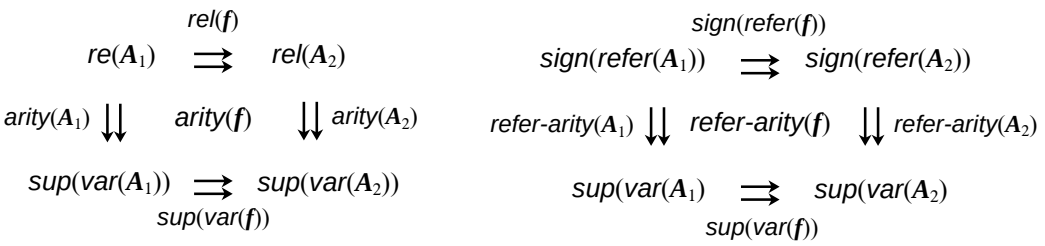


Diagram 8: Reference-arity and arity classification squares

- 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***) • *refer-arity*(***f***).

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

- For convenience of reference, we introduce additional model 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, a model morphism has the following component infomorphisms:

  – the *entity* infomorphism *ent*(*f*) = *info*(*refer*(*f*)) : *ent*($\boldsymbol{A}_1$) ⇄ *ent*($\boldsymbol{A}_2$),
  – the *relation* infomorphism *rel*(*f*) = *vert-src*(*sign*(*f*)) : *rel*($\boldsymbol{A}_1$) ⇄ *rel*($\boldsymbol{A}_2$), and
  – the *variable* invertible pair *var*(*f*) = *inv-pr*(*refer*(*f*)) : *var*($\boldsymbol{A}_1$) ⇄ *var*($\boldsymbol{A}_2$).

  Therefore, a model morphism can be displayed as in Figure 1.

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

(10) (SET.FTN$function relation)
     (= (SET.FTN$source relation) model-morphism)
     (= (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) model-morphism)
     (= (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 model morphism ***f*** : $\boldsymbol{A}_1 \to \boldsymbol{A}_2$.

  – The *universe* function

  univ(***f***) = *inst*(*ent*(***f***)) : *univ*($\boldsymbol{A}_2$) → *univ*($\boldsymbol{A}_1$)

  maps objects in the universe of $\boldsymbol{A}_2$ to objects in the universe of $\boldsymbol{A}_1$.

  The *entity* infomorphism (Figure 4) looks like

  ent(***f***) = ⟨*univ*(*f*), *typ*(*ent*(*f*))⟩ : *ent*($\boldsymbol{A}_1$) ⇄ *ent*($\boldsymbol{A}_2$),

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

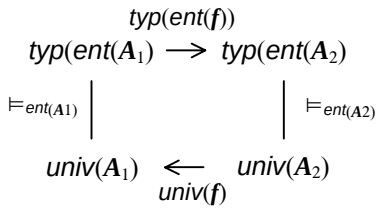


Figure 10: Entity Infomorphism

$$univ(\boldsymbol{f})(a_2) \vDash_{ent(\boldsymbol{A}1)} \alpha_1 \text{ iff } a_2 \vDash_{ent(\boldsymbol{A}2)} typ(ent(\boldsymbol{f}))(\alpha_1)$$

for all objects $a_2 \in univ(\boldsymbol{A}_2)$ and all entity types $\alpha_1 \in typ(ent(\boldsymbol{A}_1))$.

– The *tuple* function

$$tuple(\boldsymbol{f}) = inst(rel(\boldsymbol{f})) : tuple(\boldsymbol{A}_2) \to tuple(\boldsymbol{A}_1)$$

maps the relational tuples of $\boldsymbol{A}_2$ to the relational tuples of $\boldsymbol{A}_1$.

The *relation* infomorphism (Figure 5) looks like

$$rel(\boldsymbol{f}) = \langle tuple(f), typ(rel(f)) \rangle : rel(\boldsymbol{A}_1) \rightleftarrows rel(\boldsymbol{A}_2),$$

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

$$tuple(\boldsymbol{f})(r_2) \vDash_{rel(\boldsymbol{A}1)} \rho_1 \text{ iff } r_2 \vDash_{rel(\boldsymbol{A}2)} typ(rel(\boldsymbol{f}))(\rho_1)$$

for all tuples $r_2 \in tuple(\boldsymbol{A}_2)$ and all relation types $\rho_1 \in typ(rel(\boldsymbol{A}_1))$. Since the arities of both relation types and tuples are mapped by the power infomorphism of the invertible pair of $\boldsymbol{f}$, valences of both relation types and tuple are preserved and arities are in bijective correspondence with their images.

$$\begin{array}{ccc} & typ(rel(\boldsymbol{f})) & \\ typ(rel(\boldsymbol{A}_1)) & \longrightarrow & typ(rel(\boldsymbol{A}_2)) \\ \vDash_{rel(\boldsymbol{A}1)} \Big| & & \Big| \vDash_{rel(\boldsymbol{A}2)} \\ tuple(\boldsymbol{A}_1) & \underset{tuple(\boldsymbol{f})}{\longleftarrow} & tuple(\boldsymbol{A}_2) \end{array}$$

Figure 11: Relation Infomorphism

```
(12) (SET.FTN$function universe)
     (= (SET.FTN$source universe) model-morphism)
     (= (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) model-morphism)
     (= (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 model morphisms are *composable* when the target of the first is equal to the source of the second. The *composition* of two composable model morphisms $\boldsymbol{f}_1 : \boldsymbol{A} \to \boldsymbol{A}'$ and $\boldsymbol{f}_2 : \boldsymbol{A}' \to \boldsymbol{A}''$ is defined in terms of composition of their reference classification semisquares and the horizontal composition of their signature classification squares.

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

(15) (REL$relation composable)
     (= (REL$class1 composable) model-morphism)
     (= (REL$class2 composable) model-morphism)
     (= (REL$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) model-morphism)
     (forall (?f1 (model-morphism ?f1) ?f2 (model-morphism ?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)])))))
```

o Composition satisfies the usual *associative law*.

```
(forall (?f1 (model-morphism ?f1)
         ?f2 (model-morphism ?f2)
         ?f3 (model-morphism ?f3)
         (composable ?f1 ?f2) (composable ?f2 ?f3))
    (= (composition [?f1 (composition [?f2 ?f3])])
       (composition [(composition [?f1 ?f2]) ?f3])))
```

- For any model ***A***, there is an *identity* model morphism.

```
(17) (SET.FTN$function identity)
     (= (SET.FTN$source identity) mod$model)
     (= (SET.FTN$target identity) model-morphism)
     (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)))))
```

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

```
(forall (?f (model-morphism ?f))
    (and (= (composition [(identity (source ?f)) ?f]) ?f)
         (= (composition [?f (identity (target ?f))]) ?f)))
```

- A *simple* model morphism is a model morphism that has an identity variable invertible pair and an identity entity infomorphism.

```
(18) (SET$class simple)
     (SET$subclass simple model-morphism)
     (forall (?f (model-morphism ?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)))))))
```

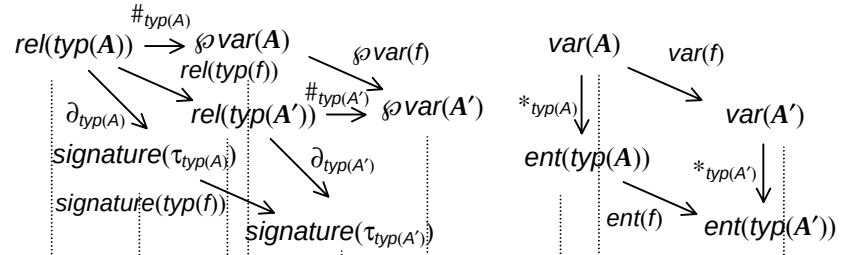


Figure 12: Type Language Morphism

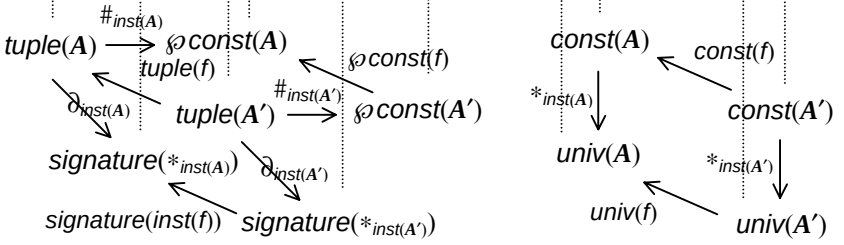


Figure 13: Instance Hypergraph Morphism

- A model morphism $\boldsymbol{f} = \langle refer(\boldsymbol{f}), sign(\boldsymbol{f}) \rangle : \boldsymbol{A} \rightleftarrows \boldsymbol{A'}$ defines two hypergraph morphisms,
  - an *instance* hypergraph morphism $inst(\boldsymbol{f}) : inst(\boldsymbol{A'}) \to inst(\boldsymbol{A})$ (Figure 12) and
  - a *type* language morphism $typ(\boldsymbol{f}) : typ(\boldsymbol{A}) \to typ(\boldsymbol{A'})$ (Figure 13).

Of course, the 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 type language morphism to a type language interpretation. The instance hypergraph morphism consists of

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

Table 5: Hypergraph Morphism Component Definitions

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

```
(19) (SET.FTN$function instance)
     (= (SET.FTN$source instance) model-morphism)
     (= (SET.FTN$target instance) hgph.mor$hypergraph-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) model-morphism)
     (= (SET.FTN$target type) lang.mor$language-morphism)
     (= (SET.FTN$composition [type lang.mor$source])
        (SET.FTN$composition [source mod$type]))
     (= (SET.FTN$composition [type lang.mor$target])
        (SET.FTN$composition [target mod$type]))
     (= (SET.FTN$composition [type lang.mor$reference])
        (SET.FTN$composition [reference cls.ssqr$type]))
     (= (SET.FTN$composition [type lang.mor$signature
        (SET.FTN$composition [signature cls.sqr$type]))
     (= (SET.FTN$composition [type lang.mor$arity])
        (SET.FTN$composition [arity cls.sqr$type]))
     (= (SET.FTN$composition [type lang.mor$variable])
        (SET.FTN$composition [variable set.invpr$direct]))
     (= (SET.FTN$composition [type lang.mor$entity])
        (SET.FTN$composition [entity cls.info$type]))
     (= (SET.FTN$composition [type lang.mor$relation])
        (SET.FTN$composition [relation cls.info$type]))
```