The IFF Algebraic Theory (meta) Ontology

A Rough Cut!

This meta-ontology has been released in order to show the full architecture of the SUO IFF. A more polished complete ontology will be released later.

The Namespace of Terms

To do list:

- In the IFF-MT, define theories – these correspond to equational presentations. Also, define theory-models and their morphisms – these correspond to varieties of algebras and their morphisms.
- Define term morphisms along a language morphism. In particular, define η & μ.
- Define the term monad and the associated Kleisli category.
- Define term functor associated with type language morphism.
- Define algebra (inverse-image) operators along a language morphism.
- Define the quotient term category for equational presentations.

For every type language morphism $\boldsymbol{f} = \langle \mathit{refer}(\boldsymbol{f}), \mathit{sign}(\boldsymbol{f}), \mathit{typ}(\boldsymbol{f}) \rangle : \boldsymbol{L}_1 \to \boldsymbol{L}_2$ define

$\mathit{case}(\boldsymbol{f}) : \mathit{case}(\boldsymbol{L}_1) \to \mathit{case}(\boldsymbol{L}_2)$

$\mathit{term}(\boldsymbol{f}) : \mathit{term}(\boldsymbol{L}_1) \to \mathit{term}(\boldsymbol{L}_2)$

$\mathit{tuple}(\boldsymbol{f}) : \mathit{tuple}(\boldsymbol{L}_1) \to \mathit{tuple}(\boldsymbol{L}_2)$

η is the atom function $\mathit{atom}(\boldsymbol{f}) : \mathit{ftn}(\boldsymbol{L}) \to \mathit{term}(\boldsymbol{L})$

This section offers an axiomatization for terms in a 1st-order language that is compatible with the IFF Model Theory Ontology (IFF-MT). Terms will be used for two purposes: (1) they will be merged into the IFF-MT, by replacing substitutions with term tuples; and they are the component of the IFF Algebraic Theory Ontology (IFF-AT) analogous to the expression component in the namespace of the IFF relation type languages. For the first purpose, term tuples subsume the previous notion of substitution. In addition, suitable modifications need to be made to the IFF-MT by replacing the effect of substitutions in language morphisms, language expressions and language colimits, with the effect of term tuples. The IFF-AT, which represents the traditional Lawverian functorial semantics generalizing universal algebra, will be functorially embedded into the IFF-MT. Thus, the Lawverian functorial semantics is a special case of the classification-oriented IFF semantics – IFF algebras are special IFF models.

Table 1 lists the terminology for function types and terms.

Table 1: Terminology for terms

	Class	Function	Other
lang	language = operator -domain	reference function-arity signature type reference-assign reference-arity variable entity **function** arity-fiber constant term	
		renaming domain codomain	
		arity case injection indication projection	
lang.mor	language- morphism	source target reference function-arity signature type reference-assign reference-arity variable entity function composition identity	composable -opspan composable
lang.term		**term** arity signature type arity-fiber index-opspan indexed-term	
		element atom substitution-opspan substitutable substitution	
		is-element = elementary is-substitution = composite	
		term-substitution-opspan term-substitutable term- substitution	
		indicia variable function tuple resolution	
lang.eqn		equation arity type term1 term2 equational-presentation language equation-set	
lang.tpl		**tuple** index arity	
		empty tuplable-opspan tuplable tuplable-cocone tupling singleton = atom	
		is-empty is-nonempty is-atom = is-singleton	
		insertible-opspan insertible insertion inclusion renamable renaming	
		index-arity case injection indication projection	
		selection remainder = rest	
		composable-opspan composable composition identity	
lang.tpl .col		initial counique	
lang.tpl .col.coprd2		diagram = pair subset1 subset2	
		renaming1 renaming2	
		cocone cocone-diagram opvertex opfirst opsecond	
		colimiting-cocone colimit = binary-coproduct injection1 injection2	
		comediator	

Membership as Coproduct

`set.mbr`

We assume that the following code appears in the IFF Lower Core Ontology in the '`set.mbr`' sub-namespace.

- The extent of the membership relation on subsets of an underlying set A is the coproduct of the power identity function, regarded as a coproduct *arity*

  $\#_{\mathbf{G}} = \mathit{id}_{\wp A} : \wp A \to \wp A.$

```
(1) (SET.FTN$function member-arity)
    (= (SET.FTN$source member-arity) set$set)
    (= (SET.FTN$target member-arity) set.col.art$arity)
    (= (SET.FTN$composition [member-arity set.col.art$index]) set$power)
    (= (SET.FTN$composition [member-arity set.col.art$base]) (SET.FTN$identity set$set))
    (= (SET.FTN$composition [member-arity set.col.art$function])
       (SET.FTN$composition [set$power set.ftn$identity]))
```

- Any set A has a set of membership *cases*

  $\mathit{mbr}(A) = \sum \mathit{id}_{\wp A}$
  $= \sum_{X \in \wp A} \mathit{id}_{\wp A}(X)$
  $= \{(X, x) \mid X \in \wp A, x \in X\},$

  that is the coproduct of its power identity function as arity.

  For any set A and any subset $X \in \wp A$ there is a member *injection* function:

  $\mathit{inj}(A)(X) : X \to \mathit{mbr}(A)$

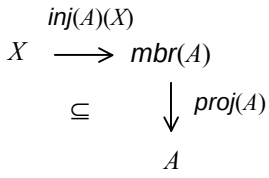

Diagram 1: Coproduct

  defined by $\mathit{inj}(A)(X)(x) = (X, x)$ for all subsets $X \in \wp A$ and all elements $x \in X$. Obviously, the injections are injective. They commute (Diagram 1) with projection and inclusion.

```
(2) (SET.FTN$function member)
    (= (SET.FTN$source member) set$set)
    (= (SET.FTN$target member) set$set)
    (= member (SET.FTN$composition [member-arity set.col.art$coproduct]))

(3) (KIF$function injection)
    (= (KIF$source injection) set$set)
    (= (KIF$target injection) SET.FTN$function)
    (= injection (SET.FTN$composition [member-arity set.col.art$injection]))
```

- Any set A defines *indication* and *projection* functions based on its arity (Figure 2):

  $\mathit{indic}(A) : \mathit{mbr}(A) \to \wp A,$
  $\mathit{proj}(A) : \mathit{mbr}(A) \to A.$

  These are defined by

  $\mathit{indic}(A)((X, x)) = X$ and $\mathit{proj}(A)((X, x)) = x$

  for all subsets $X \in \wp A$ and all elements $x \in X$.

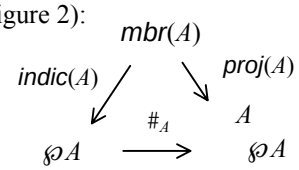

Figure 2: Indication and projection

```
(4) (SET.FTN$function indication)
    (= (SET.FTN$source indication) set$set)
    (= (SET.FTN$target indication) set.ftn$function)
    (= indication (SET.FTN$composition [member-arity set.col.art$indication]))

(5) (SET.FTN$function projection)
    (= (SET.FTN$source projection) set$set)
    (= (SET.FTN$target projection) set.ftn$function)
    (= projection (SET.FTN$composition [member-arity set.col.art$projection]))
```

Function Type Languages

`lang`

The function type languages in the IFF algebraic theory Ontology (IFF-AT) represent the operator domains of many-sorted universal algebra. Examples include monoids, groups, rings, modules, and automata.

- A 1st-order *function type language* $\boldsymbol{L} = \langle refer(\boldsymbol{L}), arity(\boldsymbol{L}), rtn(\boldsymbol{L})\rangle$ (Figure 1) consists of
  - a *reference* or sort function
    $*_{\boldsymbol{L}} = refer(\boldsymbol{L}) : var(\boldsymbol{L}) \to ent(\boldsymbol{L})$,
  - a *signature* function
    $\partial_{\boldsymbol{L}} = sign(\boldsymbol{L}) : ftn(\boldsymbol{L}) \to sign(*_{\boldsymbol{L}})$, and
  - a *return type* function
    $\tau_{\boldsymbol{L}} = typ(\boldsymbol{L}) : ftn(\boldsymbol{L}) \to ent(\boldsymbol{L})$

  that satisfy the pullback constraints
  $src(sign(\boldsymbol{L})) = src(rtn(\boldsymbol{L}))$,
  $tgt(sign(\boldsymbol{L})) = sign(refer(\boldsymbol{L}))$, and
  $tgt(rtn(\boldsymbol{L})) = tgt(refer(\boldsymbol{L}))$.

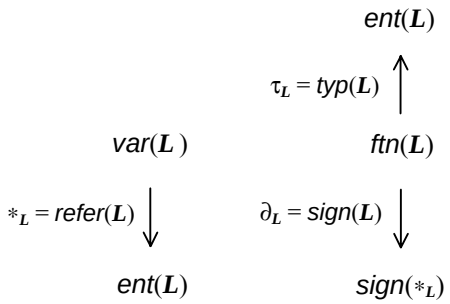

Figure 1: Function Type Language

We represent a function type $f \in ftn(\boldsymbol{L})$ having $sign(\boldsymbol{L})(f) = \tau$ and $rtn(\boldsymbol{L})(f) = \alpha$ with the linear notation $f(\tau) : \alpha$.

```
(1) (SET$class language)
    (SET$class operator-domain)
    (= operator-domain language)

(2) (SET.FTN$function reference)
    (= (SET.FTN$source reference) language)
    (= (SET.FTN$target reference) set.ftn$function)

(3) (SET.FTN$function signature)
    (= (SET.FTN$source signature) language)
    (= (SET.FTN$target signature) set.ftn$function)

(4) (SET.FTN$function type)
    (= (SET.FTN$source type) language)
    (= (SET.FTN$target type) set.ftn$function)

(5) (= (SET.FTN$composition [signature set.ftn$source])
       (SET.FTN$composition [type set.ftn$source]))

(6) (= (SET.FTN$composition [signature set.ftn$target])
       (SET.FTN$composition [reference set.ftn$signature]))

(7) (= (SET.FTN$composition [type set.ftn$target])
       (SET.FTN$composition [reference set.ftn$target]))
```

- For convenience of theoretical presentation, we introduce additional type language terminology for the composition between the reference function and the signature assign and arity functions.
  - *reference-assign* function $refer\text{-}assign(\boldsymbol{L}) = sign\text{-}assign(*_{\boldsymbol{L}}) : \wp var(\boldsymbol{L}) \to sign(refer(\boldsymbol{L}))$,
  - *reference-arity* function $refer\text{-}arity(\boldsymbol{L}) = sign\text{-}arity(*_{\boldsymbol{L}}) : sign(refer(\boldsymbol{L})) \to \wp var(\boldsymbol{L})$.

  As we know, these designations are inverse to each other:
  $refer\text{-}assign(\boldsymbol{L}) \bullet refer\text{-}arity(\boldsymbol{L}) = id_{\wp var(\boldsymbol{L})}$, and
  $refer\text{-}arity(\boldsymbol{L}) \bullet refer\text{-}assign(\boldsymbol{L}) = id_{sign(refer(\boldsymbol{L}))}$.

```
(8) (SET.FTN$function reference-assign)
    (= (SET.FTN$source reference-assign) language)
    (= (SET.FTN$target reference-assign) set.ftn$function)
    (= reference-assign (SET.FTN$composition [reference set.ftn$signature-assign]))
```

```
(9) (SET.FTN$function reference-arity)
    (= (SET.FTN$source reference-arity) language)
    (= (SET.FTN$target reference-arity) set.ftn$function)
    (= reference-arity (SET.FTN$composition [reference set.ftn$signature-arity]))
```

- The set function composition of the signature function with the reference arity function defines
  - a *function-arity* function

    $$\#_{\boldsymbol{L}} = \mathit{ftn\text{-}arity}(\boldsymbol{L}) = \mathit{sign}(\boldsymbol{L}) \cdot \mathit{refer\text{-}arity}(\boldsymbol{L}) : \mathit{ftn}(\boldsymbol{L}) \to \wp\, \mathit{var}(\boldsymbol{L})$$

    which can be used in place of the signature function. The pullback constraint takes the form

    $$\mathit{tgt}(\mathit{ftn\text{-}arity}(\boldsymbol{L})) = \wp\, \mathit{var}(\boldsymbol{L}) = \wp\, \mathit{src}(\mathit{refer}(\boldsymbol{L})).$$

    Since reference arity and reference assign are inverse functions, we can equivalently define the signature function in terms of the arity function.

```
(10) (SET.FTN$function function-arity)
     (= (SET.FTN$source function-arity) language)
     (= (SET.FTN$target function-arity) set.ftn$function)
     (forall (?l (language ?l))
         (= (function-arity ?l)
            (set.ftn$composition [(signature ?l) (reference-arity ?l)])))

(11) (forall (?l (language ?l))
         (= (signature ?l)
            (set.ftn$composition [(function-arity ?l) (reference-assign ?l)])))
```

- For convenience of practical reference, we introduce additional type language terminology for the source and target components of these functions. The source of the signature function is called the set of *function* types of $\boldsymbol{L}$ and denoted $\mathit{ftn}(\boldsymbol{L})$. The target of the reference function is called the set of *entity* types of $\boldsymbol{L}$ and denoted $\mathit{ent}(\boldsymbol{L})$. The source of the reference function is called the set of *variables* of $\boldsymbol{L}$ and denoted $\mathit{var}(\boldsymbol{L})$. In summary, we provide terminology for the following sets:
  - a set of *variables* $\mathit{var}(\boldsymbol{L})$,
  - a set of *entity types* $\mathit{ent}(\boldsymbol{L})$, and
  - a set of *function types* $\mathit{ftn}(\boldsymbol{L})$.

```
(12) (SET.FTN$function variable)
     (= (SET.FTN$source variable) language)
     (= (SET.FTN$target variable) set$set)
     (= variable (SET.FTN$composition [reference set.ftn$source]))
     (= variable (SET.FTN$composition [type set.ftn$target]))

(13) (SET.FTN$function entity)
     (= (SET.FTN$source entity) language)
     (= (SET.FTN$target entity) set$set)
     (= entity (SET.FTN$composition [reference set.ftn$target]))

(14) (SET.FTN$function function)
     (= (SET.FTN$source function) language)
     (= (SET.FTN$target function) set$set)
     (= function (SET.FTN$composition [arity set.ftn$source]))
     (= function (SET.FTN$composition [type set.ftn$source]))
```

- A *constant* is a function type whose arity is empty. In particular, the arity fiber at the empty set is called the set of constants of $\boldsymbol{L}$

  $$\mathit{const}(\boldsymbol{L}) = \mathit{arity\text{-}fiber}(\boldsymbol{L})(\varnothing).$$

```
(15) (SET.FTN$function arity-fiber)
     (= (SET.FTN$source arity-fiber) lang$language)
     (= (SET.FTN$target arity-fiber) set.ftn$function)
     (= (SET.FTN$composition [arity-fiber set.ftn$source])
        (SET.FTN$composition [lang$variable set$power]))
     (= (SET.FTN$composition [arity-fiber set.ftn$target])
        (SET.FTN$composition [function set$power]))
     (forall (?l (language ?l))
         (= (arity-fiber ?l)
            (set.ftn$fiber (arity ?l))))
```

```
(16) (KIF$function constant)
     (= (SET.FTN$source constant) lang$language)
     (= (SET.FTN$target constant) set$set)
     (forall (?l (language ?l))
         (= (constant ?l)
            ((arity-fiber ?l) set.col$initial)))
```

- For any function type language

  $\boldsymbol{L} = \langle ent(\boldsymbol{L}), ftn(\boldsymbol{L}), var(\boldsymbol{L}), *_{\boldsymbol{L}}, \#_{\boldsymbol{L}}, \partial_{\boldsymbol{L}} \rangle$,

  the associated type language of *term*s (Figure 4)

  $term(\boldsymbol{L})$

  $= \langle ent(\boldsymbol{L}), ftn(term(\boldsymbol{L})), var(\boldsymbol{L}), *_{\boldsymbol{L}}, \#_{term(\boldsymbol{L})}, \partial_{term(\boldsymbol{L})} \rangle$

  has terms as its function types with the same entity types and variables as in $\boldsymbol{L}$. The reference function is unchanged, but the arity function (and also the equivalent signature function) is an extension of the arity function of $\boldsymbol{L}$ from the function types of $\boldsymbol{L}$ to the terms of $\boldsymbol{L}$. This has a recursive definition.

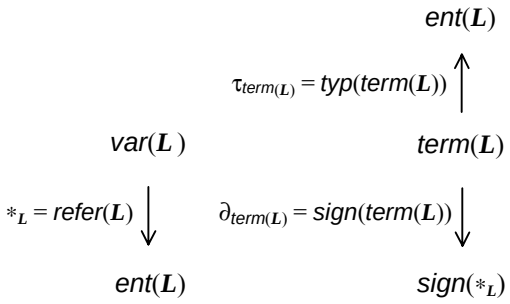

Figure 4: Term Type Language

```
(17) (SET.FTN$function term)
     (= (SET.FTN$source term) language)
     (= (SET.FTN$target term) language)
     (forall (?l (language ?l))
         (and (= (entity (term ?l))    (entity ?l))
              (= (function (term ?l))  (lang.term$term ?l))
              (= (variable (term ?l))  (variable ?l))
              (= (reference (term ?l)) (reference ?l))
              (= (arity (term ?l))     (lang.term$arity ?l))
              (= (signature (term ?l)) (lang.term$signature ?l))))
```

- For any type language $\boldsymbol{L}$ a *renaming* $h : src(h) \to tgt(h)$ of $\boldsymbol{L}$ is a renaming of the reference function. This means that it is a bijective function between variable sets $src(h), tgt(h) \subseteq var(\boldsymbol{L})$ that respects the reference function. Let $rename(\boldsymbol{L})$ denote the class of all renamings of $\boldsymbol{L}$. For convenience of reference, we rename the source and target of substitutions.

```
(18) (SET.FTN$function renaming)
     (= (SET.FTN$source renaming) language)
     (= (SET.FTN$target renaming) set$set)
     (= renaming (SET.FTN$composition [reference set.ftn$renaming]))
     (= (SET.FTN$composition [renaming set.ftn$renaming-base])
        (SET.FTN$composition [variable set$power]))

(19) (SET.FTN$function domain)
     (= (SET.FTN$source domain) language)
     (= (SET.FTN$target domain) set.ftn$function)
     (= (SET.FTN$composition [domain set.ftn$source]) renaming)
     (= domain (SET.FTN$composition [reference set.ftn$domain]))

(20) (SET.FTN$function codomain)
     (= (SET.FTN$source codomain) language)
     (= (SET.FTN$target codomain) set.ftn$function)
     (= (SET.FTN$composition [codomain set.ftn$source]) renaming)
     (= codomain (SET.FTN$composition [reference set.ftn$codomain]))
```

Example

In universal algebra, a group $\boldsymbol{G} = \langle A, \circ, (-)^{-1}, \upsilon \rangle$ consists of an *underlying set* A, a binary operation $\circ : A \times A \to A$ called the *multiplication* of $\boldsymbol{G}$, a unary operation $(-)^{-1} : A \to A$ called the *inverse* of $\boldsymbol{G}$, and a constant $\upsilon \in A$ called the *unit* of $\boldsymbol{G}$. The multiplication must satisfy the usual associative law, the unit satisfies the two left and right unit laws, and the inverse satisfies the usual two-sided inverse equations. An appropriate function type language $\boldsymbol{L}$ for groups would be have a single sort or entity type (which can be ignored) $ent(\boldsymbol{L}) = 1$, the natural numbers as variables $var(\boldsymbol{L}) = natno$, the (trivial) reference function

$$*_{\boldsymbol{L}} = refer(\boldsymbol{L}) = !_{natno} : natno \to 1,$$

the set of three function type symbols $\mathit{ftn}(\boldsymbol{L}) = \{m, i, e\}$ that denote the multiplication, inverse function, and unit of a group, respectively. These symbols have the following arities: $\mathit{ftn\text{-}arity}(\boldsymbol{L})(m) = \{0, 1\}$, $\mathit{ftn\text{-}arity}(\boldsymbol{L})(i) = \{0\}$, and $\mathit{ftn\text{-}arity}(\boldsymbol{L})(e) = \emptyset$.

Cases

- The reference arity function

  $refer\text{-}arity(\boldsymbol{L}) = sign\text{-}arity(*_{\boldsymbol{L}}) : sign(refer(\boldsymbol{L})) \to \wp\, var(\boldsymbol{L})$.

  can be regarded as a coproduct *arity*. This is the coproduct arity of the reference function.

```
(21) (SET.FTN$function arity)
     (= (SET.FTN$source arity) language)
     (= (SET.FTN$target arity) set.col.art$arity)
     (= (SET.FTN$composition [arity set.col.art$index])
        (SET.FTN$composition [reference set.ftn$signature-arity]))
     (= (SET.FTN$composition [arity set.col.art$base]) variable)
     (= (SET.FTN$composition [arity set.col.art$function]) reference-arity)
     (= arity (SET.FTN$composition [reference set.ftn$arity]))
```

- Any type language $\boldsymbol{L}$ has a set of *cases*

  $case(\boldsymbol{L}) = \sum arity(\boldsymbol{L})$
  $= \sum_{\tau \in sign(refer(\boldsymbol{L}))} arity(\boldsymbol{L})(\tau)$
  $= \{(\tau, x) \mid \tau \in sign(refer(\boldsymbol{L})), x \in arity(\boldsymbol{L})(\tau)\}$,

  that is the coproduct of its arity. It is also the case of the reference functions.

  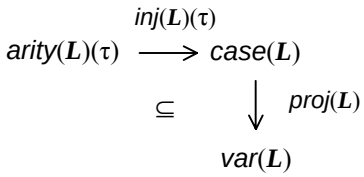

  Diagram 1: Coproduct

  For any type language $\boldsymbol{L}$ and any signature $\tau \in sign(refer(\boldsymbol{L}))$ there is an *injection* function:

  $inj(\boldsymbol{L})(\tau) : arity(\boldsymbol{L})(\tau) \to case(\boldsymbol{L})$

  defined by $inj(\boldsymbol{L})(\tau)(x) = (\tau, x)$ for all signatures $\tau \in sign(refer(\boldsymbol{L}))$ and all variables $x \in arity(\boldsymbol{L})(\tau)$. Obviously, the injections are injective. They commute (Diagram 1) with projection and inclusion.

```
(22) (SET.FTN$function case)
     (= (SET.FTN$source case) language)
     (= (SET.FTN$target case) set$set)
     (= case (SET.FTN$composition [arity set.col.art$coproduct]))
     (= case (SET.FTN$composition [reference set.ftn$case]))

(23) (KIF$function injection)
     (= (KIF$source injection) language)
     (= (KIF$target injection) SET.FTN$function)
     (= injection (SET.FTN$composition [arity set.col.art$injection]))
     (= injection (SET.FTN$composition [reference set.ftn$injection]))
```

- Any type language $\boldsymbol{L}$ defines *indication* and *projection* functions based on its reference arity (Figure 2):

  $indic(\boldsymbol{L}) : case(\boldsymbol{L}) \to sign(refer(\boldsymbol{L}))$,

  $proj(\boldsymbol{L}) : case(\boldsymbol{L}) \to var(\boldsymbol{L})$.

  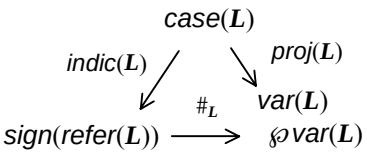

  Figure 2: Indication and projection

  These are defined by

  $indic(\boldsymbol{L})((\tau, x)) = \tau$ and $proj(\boldsymbol{L})((\tau, x)) = x$

  for all signatures $\tau \in sign(refer(\boldsymbol{L}))$ and all variables $x \in arity(\boldsymbol{L})(\tau)$.

```
(24) (SET.FTN$function indication)
     (= (SET.FTN$source indication) language)
     (= (SET.FTN$target indication) set.ftn$function)
     (= indication (SET.FTN$composition [arity set.col.art$indication]))
     (= indication (SET.FTN$composition [reference set.ftn$indication]))

(25) (SET.FTN$function projection)
     (= (SET.FTN$source projection) language)
     (= (SET.FTN$target projection) set.ftn$function)
     (= projection (SET.FTN$composition [arity set.col.art$projection]))
     (= projection (SET.FTN$composition [reference set.ftn$projection]))
```

- Any type language $\boldsymbol{L}$ defines a *comediator* function:

  $\tilde{*}_{\boldsymbol{L}} = comed(\boldsymbol{L}) : case(\boldsymbol{L}) \to ent(\boldsymbol{L})$.

This function is the slot-filler function for frames. Pointwise, it is defined by

$$comed(\boldsymbol{L})((\tau, x)) = \tau(x)$$

for all signatures $\tau \in sign(refer(\boldsymbol{L}))$ and all variables $x \in arity(\boldsymbol{L})(\tau)$.

The comediator function commutes (Diagram 4) with the case injection function and the signature itself. Since all the signatures factor as the composition of their arity inclusion with the reference function, the comediator function can simply be defined as the composition of projection with the original (reference) function. If the indexed names in $case(\boldsymbol{L})$ are viewed as roles (prehensions), the comediator is a reference function from roles to objects (actualities).

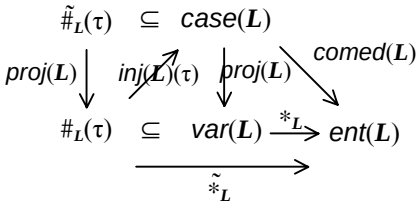

Diagram 4: Comediator

```
(26) (SET.FTN$function comediator)
     (= (SET.FTN$source comediator) language)
     (= (SET.FTN$target comediator) set.ftn$function)
     (= (SET.FTN$composition [comediator set.ftn$source]) case)
     (= (SET.FTN$composition [comediator set.ftn$target]) entity)
     (= comediator (SET.FTN$composition [reference set.ftn$comediator]))
```

Function Type Language Morphisms

`lang.mor`

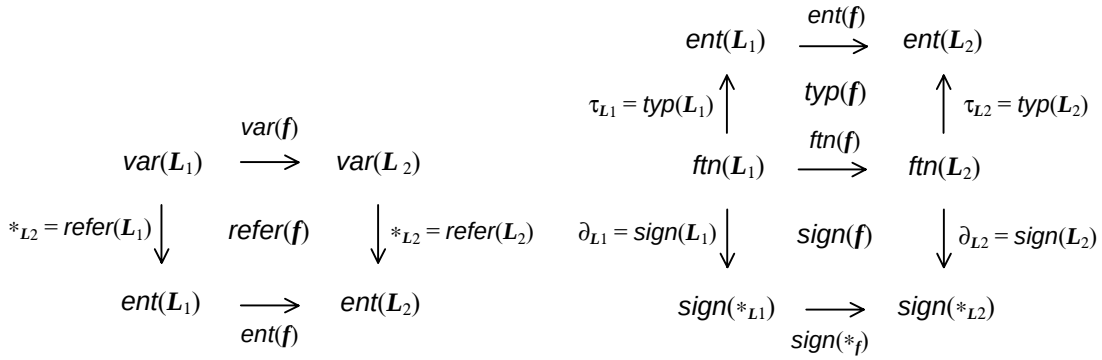

Figure 2: Function Type Language Morphism

- A 1st-order *function type language morphism* $\boldsymbol{f} = \langle refer(\boldsymbol{f}), sign(\boldsymbol{f}), typ(\boldsymbol{f}) \rangle : \boldsymbol{L}_1 \to \boldsymbol{L}_2$ from function type language $\boldsymbol{L}_1$ to function type language $\boldsymbol{L}_2$ (Figure 2) is a three dimensional construction consisting of a reference quartet $refer(\boldsymbol{f})$, a signature quartet $sign(\boldsymbol{f})$, and a return type quartet $typ(\boldsymbol{f})$, where the vertical source of the signature quartet is the vertical source of the return type quartet

  $vert\text{-}src(signature(\boldsymbol{f})) = vert\text{-}src(typ(\boldsymbol{f}))$,

  the signature of the reference quartet is the vertical target of the signature quartet

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

  and the vertical target of the reference quartet is the vertical target of the return type quartet

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

```
(1) (SET$class language-morphism)

(2) (SET.FTN$function source)
    (= (SET.FTN$source source) language-morphism)
    (= (SET.FTN$target source) lang$language)

(3) (SET.FTN$function target)
    (= (SET.FTN$source target) language-morphism)
    (= (SET.FTN$target target) lang$language)

(4) (SET.FTN$function reference)
    (= (SET.FTN$source reference) language-morphism)
    (= (SET.FTN$target reference) set.qtt$quartet)
    (= (SET.FTN$composition [reference set.qtt$horizontal-source])
       (SET.FTN$composition [source lang$reference]))
    (= (SET.FTN$composition [reference set.qtt$horizontal-target])
       (SET.FTN$composition [target lang$reference]))

(5) (SET.FTN$function signature)
    (= (SET.FTN$source signature) language-morphism)
    (= (SET.FTN$target signature) set.qtt$quartet)
    (= (SET.FTN$composition [signature set.qtt$horizontal-source])
       (SET.FTN$composition [source lang$signature]))
    (= (SET.FTN$composition [signature set.qtt$horizontal-target])
       (SET.FTN$composition [target lang$signature]))

(6) (SET.FTN$function type)
    (= (SET.FTN$source type) language-morphism)
    (= (SET.FTN$target type) set.qtt$quartet)
    (= (SET.FTN$composition [return set.qtt$horizontal-source])
       (SET.FTN$composition [source lang$type]))
```

```
      (= (SET.FTN$composition [return set.qtt$horizontal-target])
         (SET.FTN$composition [target lang$type]))

(7)  (= (SET.FTN$composition [signature set.qtt$vertical-source])
         (SET.FTN$composition [type set.qtt$vertical-source]))

(8)  (= (SET.FTN$composition [reference set.qtt$signature])
         (SET.FTN$composition [signature set.qtt$vertical-target]))

(9)  (= (SET.FTN$composition [reference set.qtt$vertical-target])
         (SET.FTN$composition [type set.qtt$vertical-target]))
```

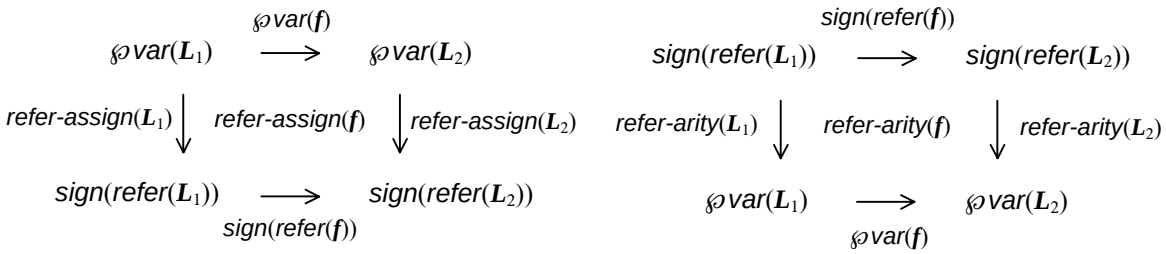

Figure 8: The reference-assign and reference-arity quartets

○ For convenience of theoretical presentation, we introduce additional type language terminology for the composition between the reference function and the signature assign and arity functions. Associated with a type language morphism $\boldsymbol{f} : \boldsymbol{L}_1 \to \boldsymbol{L}_2$ is a *reference-assign* quartet *refer-assign*(***f***) and a *reference-arity* quartet *refer-arity*(***f***).

```
(10) (SET.FTN$function reference-assign)
     (= (SET.FTN$source reference-assign) language-morphism)
     (= (SET.FTN$target reference-assign) set.qtt$quartet)
     (= reference-assign (SET.FTN$composition [reference set.qtt$signature-assign]))

(11) (SET.FTN$function reference-arity)
     (= (SET.FTN$source reference-arity) language-morphism)
     (= (SET.FTN$target reference-arity) set.qtt$quartet)
     (= reference-arity (SET.FTN$composition [reference set.qtt$signature-arity]))
```

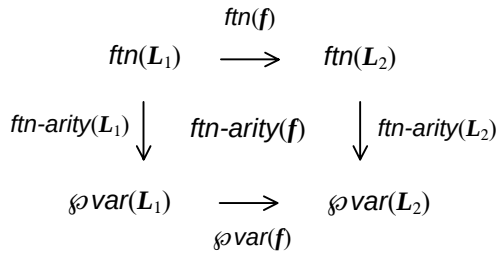

Figure 9: The function-arity quartet

○ The vertical composition of the signature quartet with the reference arity quartet defines a *function-arity* quartet $\#_{\boldsymbol{L}}$ = *ftn-arity*(***L***) = *sign*(***L***) • *refer-arity*(***L***) (Figure 9), which can be used in place of the signature quartet. The pullback constraint takes the form *vert-tgt*(*ftn-arity*(***f***)) = ℘*vert-src*(*refer*(***f***)). Since reference arity and reference assign are (vertically) inverse quartets, we can equivalently define the signature function in terms of the function-arity function.

```
(12) (SET.FTN$function function-arity)
     (= (SET.FTN$source function-arity) language-morphism)
```

```
      (= (SET.FTN$target function-arity) set.qtt$quartet)
      (forall (?f (language-morphism ?f))
           (= (function-arity ?f)
              (set.qtt$vertical-composition [(signature ?f) (reference-arity ?f)])))

(13) (forall (?f (language-morphism ?f))
         (= (signature ?f)
            (set.qtt$vertical-composition
                [(function-arity ?f) (reference-assign ?f)])))
```

- For convenience of reference, we introduce additional type language terminology for the vertical source and target components of these quartets. The vertical source of the reference quartet is called the *variable* function of $\boldsymbol{f}$ and denoted $\mathit{var}(\boldsymbol{f})$. The vertical target of the reference and return type quartets is called the *entity* type function of $\boldsymbol{f}$ and denoted $\mathit{ent}(\boldsymbol{f})$. The vertical source of the signature and return type quartets is called the *function* type function of $\boldsymbol{f}$ and denoted $\mathit{typ}(\boldsymbol{f})$. In summary, a type language morphism has the following component functions:
  - the *variable* function $\mathit{var}(\boldsymbol{f}) = \mathit{vert\text{-}src}(\mathit{refer}(\boldsymbol{f})) : \mathit{var}(\boldsymbol{L}_1) \to \mathit{var}(\boldsymbol{L}_2)$,
  - the *entity* type function $\mathit{ent}(\boldsymbol{f}) = \mathit{vert\text{-}tgt}(\mathit{refer}(\boldsymbol{f})) : \mathit{ent}(\boldsymbol{L}_1) \to \mathit{ent}(\boldsymbol{L}_2)$, and
  - the *function* type function $\mathit{ftn}(\boldsymbol{f}) = \mathit{vert\text{-}src}(\mathit{sign}(\boldsymbol{f})) : \mathit{ftn}(\boldsymbol{L}_1) \to \mathit{ftn}(\boldsymbol{L}_2)$.

  A type language morphism is determined by the variable, entity type and function type functions, and alternatively can be expressed and symbolized as

  $$\boldsymbol{f} = \langle \mathit{var}(\boldsymbol{f}), \mathit{ent}(\boldsymbol{f}), \mathit{ftn}(\boldsymbol{f}) \rangle : \boldsymbol{L}_1 \to \boldsymbol{L}_2.$$

```
(14) (SET.FTN$function variable)
     (= (SET.FTN$source variable) language-morphism)
     (= (SET.FTN$target variable) set.ftn$function)
     (= variable (SET.FTN$composition [reference set.qtt$vertical-source]))

(15) (SET.FTN$function entity)
     (= (SET.FTN$source entity) language-morphism)
     (= (SET.FTN$target entity) set.ftn$function)
     (= entity (SET.FTN$composition [reference set.qtt$vertical-target]))
     (= entity (SET.FTN$composition [type set.qtt$vertical-target]))

(16) (SET.FTN$function function)
     (= (SET.FTN$source function) language-morphism)
     (= (SET.FTN$target function) set.ftn$function)
     (= function (SET.FTN$composition [signature set.qtt$vertical-source]))
     (= function (SET.FTN$composition [type set.qtt$vertical-source]))
```

- Two type language morphisms are *composable* when the target of the first is equal to the source of the second. The *composition* of two composable type language morphisms $\boldsymbol{f}_1 : \boldsymbol{L} \to \boldsymbol{L}'$ and $\boldsymbol{f}_2 : \boldsymbol{L}' \to \boldsymbol{L}''$ is defined in terms of the horizontal composition of their reference, signature and return type quartets.

```
(17) (SET.LIM.PBK$opspan composable-opspan)
     (= (SET.LIM.PBK$class1 composable-opspan) language-morphism)
     (= (SET.LIM.PBK$class2 composable-opspan) language-morphism)
     (= (SET.LIM.PBK$opvertex composable-opspan) lang$language)
     (= (SET.LIM.PBK$first composable-opspan) target)
     (= (SET.LIM.PBK$second composable-opspan) source)

(18) (REL$relation composable)
     (= (REL$class1 composable) language-morphism)
     (= (REL$class2 composable) language-morphism)
     (= (REL$extent composable) (SET.LIM.PBK$pullback composable-opspan))

(19) (SET.FTN$function composition)
     (= (SET.FTN$source composition) (SET.LIM.PBK$pullback composable-opspan))
     (= (SET.FTN$target composition) language-morphism)
     (forall (?f1 (language-morphism ?f1)
              ?f2 (language-morphism ?f2)
              (composable ?f1 ?f2))
         (and (= (source (composition [?f1 ?f2])) (source ?f1))
              (= (target (composition [?f1 ?f2])) (target ?f2))
              (= (reference (composition [?f1 ?f2]))
                 (set.qtt$horizontal-composition [(reference ?f1) (reference ?f2)]))
```

```
            (= (signature (composition [?f1 ?f2]))
               (set.qtt$horizontal-composition [(signature ?f1) (signature ?f2)]))
            (= (type (composition [?f1 ?f2]))
               (set.qtt$horizontal-composition [(type ?f1) (type ?f2)])))
```

- Composition satisfies the usual *associative law*.

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

- For any type language ***L***, there is an *identity* type language morphism.

```
(20) (SET.FTN$function identity)
     (= (SET.FTN$source identity) lang$language)
     (= (SET.FTN$target identity) language-morphism)
     (forall (?l (lang$language ?l))
         (and (= (source (identity ?l)) ?l)
              (= (target (identity ?l)) ?l)
              (= (reference (identity ?l))
                 (set.qtt$horizontal-identity (lang$reference ?l)))
              (= (signature (identity ?l))
                 (set.qtt$horizontal-identity (lang$signature ?l)))
              (= (type (identity ?l))
                 (set.qtt$horizontal-identity (lang$type ?l)))))
```

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

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

Cases

Terms

`lang.term`

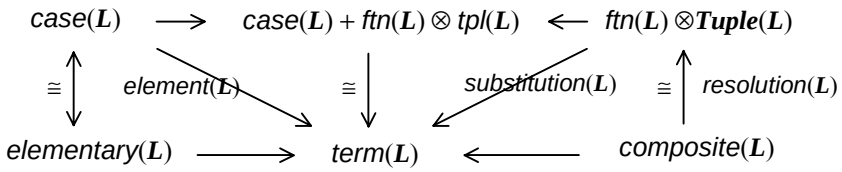

Diagram 1: The Term-Tuple Datatype as a Fixpoint Solution

This section and its term tuple subsection axiomatize the terms for a function type language $\boldsymbol{L}$. Terms and term tuples are corecursively defined. This specification replaces the traditional recursive tree-forest set equations

$\boldsymbol{Tree}(\boldsymbol{A}) \cong \mathbf{1} + \boldsymbol{A} \times \boldsymbol{Forest}(\boldsymbol{A})$,

$\boldsymbol{Forest}(\boldsymbol{A}) \cong \mathit{stack}(\boldsymbol{Tree}(\boldsymbol{A}))$

with the recursive term-tuple set equations

$\boldsymbol{Term}(\boldsymbol{L}) \cong \mathit{case}(\boldsymbol{L}) + \mathit{ftn}(\boldsymbol{L}) \otimes \boldsymbol{Tuple}(\boldsymbol{L})$,

$\boldsymbol{Tuple}(\boldsymbol{L}) \cong \mathit{tuple}(\boldsymbol{Term}(\boldsymbol{L}))$,

where '⊗', which denotes the substitution operator, will be generalized in the section on term tuples to composition in a Kleisli-like term category. Categorically, the terms-tuples set pair is the fixpoint solution

$\langle \boldsymbol{Term}(\boldsymbol{L}), \boldsymbol{Tuple}(\boldsymbol{L}) \rangle \cong F_{\mathit{term}(\boldsymbol{L})}(\langle \boldsymbol{Term}(\boldsymbol{L}), \boldsymbol{Tuple}(\boldsymbol{L}) \rangle)$,

for the $\boldsymbol{\omega}$-continuous endofunctor $F_{\mathit{term}(\boldsymbol{L})} : \mathsf{Set} \times \mathsf{Set} \to \mathsf{Set} \times \mathsf{Set}$ on the category $\mathsf{Set} \times \mathsf{Set}$ defined by

$F_{\mathit{term}(\boldsymbol{L})}(\langle \boldsymbol{X}, \boldsymbol{Y} \rangle) = \langle \mathit{case}(\boldsymbol{L}) + \mathit{ftn}(\boldsymbol{L}) \otimes \boldsymbol{Y}, \mathit{tuple}(\boldsymbol{X}) \rangle$.

- For any type language $\boldsymbol{L}$, we give a pointwise recursive definition for the set of $\boldsymbol{L}$-terms $\mathit{term}(\boldsymbol{L})$. Like any other recursively defined datatype, beyond the *attribute* functions that describe terms, there will be *basic constructor* functions, *composite constructor* functions, *Boolean tests* for the kinds of terms built by the constructors, and *selector* functions that are generalized inverses to the basic constructors.

  A term $\alpha \in \mathit{term}(\boldsymbol{L})$ will have an *signature* $\mathit{sign}(\boldsymbol{L})(\alpha)$ and a return *type* $\mathit{typ}(\boldsymbol{L})(\alpha)$. Hence, there is

  – a *signature* function

  $\partial_{\mathit{term}(\boldsymbol{L})} = \mathit{sign}(\mathit{term}(\boldsymbol{L})) : \mathit{term}(\boldsymbol{L}) \to \mathit{sign}(*_{\boldsymbol{L}})$, and

  – an *arity* function

  $\#_{\mathit{expr}(\boldsymbol{L})} = \mathit{arity}(\mathit{term}(\boldsymbol{L})) = \mathit{sign}(\mathit{term}(\boldsymbol{L})) \cdot \mathit{refer\text{-}arity}(\boldsymbol{L}) : \mathit{term}(\boldsymbol{L}) \to \wp \mathit{var}(\boldsymbol{L})$

  – a *return type* function

  $\tau_{\mathit{term}(\boldsymbol{L})} = \mathit{typ}(\mathit{term}(\boldsymbol{L})) : \mathit{term}(\boldsymbol{L}) \to \mathit{ent}(\boldsymbol{L})$

  that satisfy the pullback constraints

  $\mathit{src}(\mathit{sign}(\mathit{term}(\boldsymbol{L}))) = \mathit{src}(\mathit{typ}(\mathit{term}(\boldsymbol{L})))$,

  $\mathit{tgt}(\mathit{sign}(\mathit{term}(\boldsymbol{L}))) = \mathit{sign}(\mathit{refer}(\boldsymbol{L}))$, and

  $\mathit{tgt}(\mathit{typ}(\mathit{term}(\boldsymbol{L}))) = \mathit{tgt}(\mathit{refer}(\boldsymbol{L}))$.

  

  Figure 3: A term *t*

  We picture a term α as in Figure 3, and use the linear notation

  (signature) $t : \tau \rhd \alpha$ <u>or</u> (arity) $t : A \rhd \alpha$

  to denote the term t with $\mathit{sign}(\boldsymbol{L})(t) = \tau$, $\mathit{arity}(\boldsymbol{L})(t) = A$ and $\mathit{typ}(\boldsymbol{L})(t) = \alpha$.

```
(1) (SET.FTN$function term)
    (= (SET.FTN$source term) lang$language)
    (= (SET.FTN$target term) set$set)

(2) (SET.FTN$function signature)
```

```
    (= (SET.FTN$source signature) language)
    (= (SET.FTN$target signature) set.ftn$function)
    (= (SET.FTN$composition [signature set.ftn$source]) term)
    (= (SET.FTN$composition [signature set.ftn$target])
       (SET.FTN$composition [lang$reference set.ftn$signature]))

(3) (SET.FTN$function arity)
    (= (SET.FTN$source arity) lang$language)
    (= (SET.FTN$target arity) set.ftn$function)
    (= (SET.FTN$composition [arity set.ftn$source]) term)
    (= (SET.FTN$composition [arity set.ftn$target])
       (SET.FTN$composition [lang$variable set$power]))

(4) (SET.FTN$function type)
    (= (SET.FTN$source type) language)
    (= (SET.FTN$target type) set.ftn$function)
    (= (SET.FTN$composition [type set.ftn$source]) term)
    (= (SET.FTN$composition [type set.ftn$target]) lang$entity)
```

- The arity and signature functions are equivalent, and can be expressed in terms of each other with the reference arity and reference signature inverse functions:

  $\partial_{expr(\boldsymbol{L})} = \mathit{sign}(\mathit{expr}(\boldsymbol{L})) = \mathit{arity}(\boldsymbol{L}) \cdot \mathit{refer\text{-}assign}(\boldsymbol{L}) : \mathit{term}(\boldsymbol{L}) \to \mathit{sign}(\mathit{refer}(\boldsymbol{L}))$ and

  $\#_{expr(\boldsymbol{L})} = \mathit{arity}(\mathit{term}(\boldsymbol{L})) = \mathit{sign}(\boldsymbol{L}) \cdot \mathit{refer\text{-}arity}(\boldsymbol{L}) : \mathit{term}(\boldsymbol{L}) \to \wp\mathit{var}(\boldsymbol{L})$,

  where $\mathit{refer\text{-}arity}(\boldsymbol{L}) = \mathit{sign\text{-}arity}(\tau_{\boldsymbol{L}}) : \mathit{sign}(\mathit{refer}(\boldsymbol{L})) \to \wp\mathit{var}(\boldsymbol{L})$ is the reference-arity function for language $\boldsymbol{L}$.

```
(5) (forall (?l (language ?l))
        (= (arity ?l)
           (set.ftn$composition [(signature ?l) (lang$reference-arity ?l)])))

(6) (forall (?l (language ?l))
        (= (signature ?l)
           (set.ftn$composition [(arity ?l) (lang$reference-assign ?l)])))
```

- These functions clearly satisfy the following constraints. This verifies that any function type language has an associated term type language.

```
(7) (= (SET.FTN$composition [signature set.ftn$source])
       (SET.FTN$composition [type set.ftn$source]))

(8) (= (SET.FTN$composition [type set.ftn$target])
       (SET.FTN$composition [lang$reference set.ftn$target]))
```

- It is sometime useful to reference the *fibers* of the term *arity* function.

```
(9) (SET.FTN$function arity-fiber)
    (= (SET.FTN$source arity-fiber) lang$language)
    (= (SET.FTN$target arity-fiber) set.ftn$function)
    (= (SET.FTN$composition [arity-fiber set.ftn$source])
       (SET.FTN$composition [lang$variable set$power]))
    (= (SET.FTN$composition [arity-fiber set.ftn$target])
       (SET.FTN$composition [term set$power]))
    (forall (?l (language ?l))
        (= (arity-fiber ?l)
           (set.ftn$fiber (arity ?l))))
```

- An *indexed term* (x, t) consists of a variable $x \in \mathit{var}(\boldsymbol{L})$ and a term

  $t : A \rhd \alpha$

  with $\mathit{refer}(\boldsymbol{L})(x) = \alpha = \mathit{typ}(\boldsymbol{L})(t)$.

```
(10) (SET.FTN$function index-opspan)
     (= (SET.FTN$source index-opspan) lang$language)
     (= (SET.FTN$target index-opspan) set.lim.pbk$opspan)
     (= (SET.FTN$composition [index-opspan set.lim.pbk$set1]) lang$variable)
     (= (SET.FTN$composition [index-opspan set.lim.pbk$set2]) term)
     (= (SET.FTN$composition [index-opspan set.lim.pbk$opvertex]) lang$entity)
     (= (SET.FTN$composition [index-opspan set.lim.pbk$opfirst]) lang$reference)
     (= (SET.FTN$composition [index-opspan set.lim.pbk$opsecond]) type)
```

```
(11) (SET.FTN$function indexed-term)
     (= (SET.FTN$source indexed-term) lang$language)
     (= (SET.FTN$target indexed-term) rel$relation)
     (= indexed-term (SET.FTN$composition [index-opspan set.lim.pbk$relation]))
```

Constructors

The following two primitive constructors correspond to the two summands in the equation. Their axiomatization is base upon the properties of the term-tuple datatype as the fixpoint solution of the pair of mutually recursive set equations.

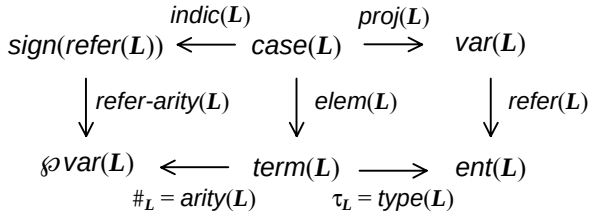

Figure 2: Elementary terms

- Variables are terms. A case (τ, x) consisting of an indexing signature $\tau \in \mathit{sign}(*_{\boldsymbol{L}})$ and a variable $x \in \mathit{arity}(\boldsymbol{L})(\tau)$ that references an entity type $\mathit{refer}(\boldsymbol{L})(x) = \alpha$, defines an *element* or *variable* term

  $$\in_{\tau, x} : \tau \rhd \alpha$$

  with $\mathit{sign}(\boldsymbol{L})(\in_{\tau, x}) = \tau$ and $\mathit{typ}(\boldsymbol{L})(\in_{\tau, x}) = \alpha$. A term is *elementary* when it is constructed from an indexed variable.

- Term tuples can be substituted into functions. A function type $f \in \mathit{ftn}(\boldsymbol{L})$ and a term tuple $\varphi \in \mathit{term}^*(\boldsymbol{L})$ are a *substitutable* pair (f, φ) when the function arity of f is the same as the index of the term tuple $\varphi : \mathit{arity}(\boldsymbol{L})(f) = \mathit{index}(\boldsymbol{L})(\varphi)$. For any substitutable pair

  $$f(B) : \alpha \text{ and } \varphi : B \to A$$

  there is a *substitution* term

  $$f[\varphi] : \mathit{refer\text{-}sign}(\boldsymbol{L})(A) \rhd \alpha$$

  with $\mathit{sign}(\boldsymbol{L})(f[\varphi]) = \mathit{refer\text{-}sign}(\boldsymbol{L})(\mathit{arity}(\boldsymbol{L})(\alpha))$ and $\mathit{typ}(\boldsymbol{L})(f[\varphi]) = \mathit{typ}(\boldsymbol{L})(f) = \alpha$. A term is *composite* when it is constructed by substitution.

  Function types are terms. As a special case, by substituting into a function the identity term on the function arity, the function $f(\tau) : \alpha$ becomes an *atomic* term $f : \tau \rhd a$.

```
(12) (SET.FTN$function element)
     (= (SET.FTN$source element) language)
     (= (SET.FTN$target element) set.ftn$function)
     (= (SET.FTN$composition [element set.ftn$source]) lang$case)
     (= (SET.FTN$composition [element set.ftn$target]) term)
     (forall (?l (lang$language ?l))
         (and (= (set.ftn$composition [(element ?l) (arity ?l)])
                 (set.ftn$composition [(lang$indication ?l) (lang$reference-arity ?l)]))
              (= (set.ftn$composition [(element ?l) (type ?l)])
                 (set.ftn$composition [(lang$projection ?l) (lang$reference ?l)])))))

(13) (SET.FTN$function substitution-opspan)
     (= (SET.FTN$source substitution-opspan) lang$language)
     (= (SET.FTN$target substitution-opspan) set.lim.pbk$opspan)
     (= (SET.FTN$composition [substitution-opspan set.lim.pbk$set1]) lang$function)
     (= (SET.FTN$composition [substitution-opspan set.lim.pbk$set2]) lang.tpl$tuple)
     (= (SET.FTN$composition [substitution-opspan set.lim.pbk$opvertex])
        (SET.FTN$composition [lang$variable set$power]))
     (= (SET.FTN$composition [substitution-opspan set.lim.pbk$opfirst]) lang$function-arity)
     (= (SET.FTN$composition [substitution-opspan set.lim.pbk$opsecond]) lang.tpl$index)

(14) (SET.FTN$function substitutable)
     (= (SET.FTN$source substitutable) lang$language)
     (= (SET.FTN$target substitutable) rel$relation)
     (= substitutable (SET.FTN$composition [substitution-opspan set.lim.pbk$relation]))

(15) (SET.FTN$function substitution)
     (= (SET.FTN$source substitution) lang$language)
     (= (SET.FTN$target substitution) set.ftn$function)
```

```
        (= (SET.FTN$composition [substitution set.ftn$source])
           (SET.FTN$composition [substitutable rel$extent]))
        (= (SET.FTN$composition [substitution set.ftn$target]) term)
        (forall (?l (lang$language ?l))
            (and (= (set.ftn$composition [(substitution ?l) (arity ?l)])
                    (set.ftn$composition
                        [(rel$second (substitutable ?l)) (lang.tpl$arity ?l)]))
                 (= (set.ftn$composition [(substitution ?l) (type ?l)])
                    (set.ftn$composition [(rel$first (substitutable ?l)) (lang$type ?l)])))))

   (16) (SET.FTN$function atom)
        (= (SET.FTN$source atom) lang$language)
        (= (SET.FTN$target atom) set.ftn$function)
        (= (SET.FTN$composition [atom set.ftn$source]) lang$function)
        (= (SET.FTN$composition [atom set.ftn$target]) term)
        (forall (?l (lang$language ?l))
            (and (= (set.ftn$composition [(atom ?l) (arity ?l)]) (lang$function-arity ?l))
                 (= (set.ftn$composition [(atom ?l) (type ?l)]) (lang$type ?l))
                 (forall (?f ((lang$function ?l) ?f)
                     (= ((atom ?l) ?f)
                        ((substitution ?l)
                            [?f (lang.tpl$identity ((lang$function-arity ?l) ?f))])))))
```

Booleans

A term is elementary when it is built by the element constructor. A term is composite when it is built by the substitution constructor. A term must be either elementary or composite – the elementary and composite terms comprise a partition for the set of terms. These Boolean tests are used to define composite constructors and the domain of selectors.

```
(17) (SET.FTN$function is-element)
     (SET.FTN$function elementary)
     (= elementary is-element)
     (= (SET.FTN$source is-element) language)
     (= (SET.FTN$target is-element) set$set)
     (forall (?l (lang$language ?l))
         (and (set$subset (is-element ?l) (term ?l))
              (= (is-element ?l) (set.ftn$image (element ?l)))))

(18) (SET.FTN$function is-substitution)
     (SET.FTN$function composite)
     (= composite is-substitution)
     (= (SET.FTN$source is-substitution) lang$language)
     (= (SET.FTN$target is-substitution) set$set)
     (forall (?l (lang$language ?l))
         (and (set$subset (is-substitution ?l) (term ?l))
              (= (is-substitution ?l) (set.ftn$image (substitution ?l)))))

(19) (forall (?l (lang$language ?l))
         (and (= (set$binary-intersection [(elementary ?l) (composite ?l)]) set.col$initial)
              (= (set$binary-union [(elementary ?l) (composite ?l)]) (term ?l))))
```

Composite Constructors

- Term tuples can be substituted into terms. A term $t \in \mathit{term}(\boldsymbol{L})$ and a term tuple $\varphi \in \mathit{term}^*(\boldsymbol{L})$ are a *term-substitutable* pair when the arity of t is the same as the index of φ : $\mathit{arity}(\boldsymbol{L})(t) = \mathit{index}(\boldsymbol{L})(\varphi)$. For any term-substitutable pair

  $t : B \rhd \alpha$ and $\varphi : B \rightharpoonup A$

  there is a *substitution* term

  $t[\varphi] : A \rhd \alpha$

  with $\mathit{arity}(\boldsymbol{L})(t[\varphi]) = \mathit{arity}(\boldsymbol{L})(\varphi) = A$ and $\mathit{rtn}(\boldsymbol{L})(t[\varphi]) = \mathit{typ}(\boldsymbol{L})(t) = b$.

  – If the term t is an elementary term

    $t = \in_{\tau, x} : \tau \rhd \alpha$ with $\mathit{arity}(\boldsymbol{L})(\tau) = B$ and $\mathit{refer}(\boldsymbol{L})(x) = \alpha$

    then the term substitution $t[\varphi]$ selects the x^{th} component term

    $t[\varphi] = \sigma_x(\varphi) : A \rhd \alpha$.

  – If the term t is an composite term

    $t = g[\psi] : B \rhd \alpha$ for some substitutable pair $g(C) : \alpha$ and $\psi : C \rightharpoonup B$

    then the term substitution resolves as tuple composition $\psi \circ \varphi$ of the two composable term tuples ψ and φ followed by substitution of the tuple composite into the function g

    $t[\varphi] = g[\psi \circ \varphi] : A \rhd \alpha$.

```
(20) (SET.FTN$function term-substitution-opspan)
     (= (SET.FTN$source term-substitution-opspan) lang$language)
     (= (SET.FTN$target term-substitution-opspan) set.lim.pbk$opspan)
     (= (SET.FTN$composition [term-substitution-opspan set.lim.pbk$set1]) term)
     (= (SET.FTN$composition [term-substitution-opspan set.lim.pbk$set2]) lang.tpl$tuple)
     (= (SET.FTN$composition [term-substitution-opspan set.lim.pbk$opvertex])
        (SET.FTN$composition [lang$variable set$power]))
     (= (SET.FTN$composition [term-substitution-opspan set.lim.pbk$opfirst]) arity)
     (= (SET.FTN$composition [term-substitution-opspan set.lim.pbk$opsecond]) lang.tpl$index)

(21) (SET.FTN$function term-substitutable)
     (= (SET.FTN$source term-substitutable) lang$language)
     (= (SET.FTN$target term-substitutable) rel$relation)
     (= term-substitutable (SET.FTN$composition [term-substitution-opspan set.lim.pbk$relation]))

(22) (SET.FTN$function term-substitution)
     (= (SET.FTN$source term-substitution) lang$language)
     (= (SET.FTN$target term-substitution) set.ftn$function)
     (= (SET.FTN$composition [term-substitution set.ftn$source])
        (SET.FTN$composition [term-substitutable rel$extent]))
     (= (SET.FTN$composition [term-substitution set.ftn$target]) term)
     (forall (?l (lang$language ?l))
         (and (= (set.ftn$composition [(term-substitution ?l) (arity ?l)])
                 (set.ftn$composition
                     [(rel$second (term-substitutable ?l)) (lang.tpl$arity ?l)]))
              (= (set.ftn$composition [(term-substitution ?l) (type ?l)])
                 (set.ftn$composition
                     [(rel$first (term-substitutable ?l)) (type ?l)])))))
     (forall (?l (lang$language ?l))
             ?a (lang.tpl$tuple ?a))
         (and (=> (exists (?t (= ((lang$reference-arity ?l) ?t) ((lang.tpl$index ?l) ?a))
                                 ?x (((lang$reference-arity ?l) ?t) ?x))
                          (= ((term-substitution ?l) [((element ?l) [?t ?x]) ?a])
                             ((lang.tpl$selection ?l) [?a ?x]))))
              (=> (exists (?g (lang$function ?g) ?b (lang.tem.tpl$tuple ?b)
                              (substitutable ?g ?b) (lang.tpl$composable ?b ?a))
                          (= ((term-substitution ?l) [((substitution ?l) [?g ?b]) ?a])
                             ((substitution ?l) [?g ((lang.tpl$composition ?l) [?b ?a])])))))))
```

Selectors

- Here we define two elementary selectors, *indicia* and *variable*, two composite selectors, *function* and *tuple*, and a combined selector called *resolution*. For elementary terms the indicia and variable selectors return the components used for the element constructor. For composite terms the function and tuple selectors return the components used for the substitution constructor. The resolution selector is the pairing of these two.

```
(23) (SET.FTN$function indicia)
     (= (SET.FTN$source indicia) lang$language)
     (= (SET.FTN$target indicia) set.ftn$function)
     (= (SET.FTN$composition [indicia set.ftn$source]) elementary)
     (= (SET.FTN$composition [indicia set.ftn$target])
        (SET.FTN$composition [lang$variable set$power]))
     (forall (?l (lang$language ?l))
         (= (set.ftn$composition [(element ?l) (indicia ?l)])
            (set.ftn$composition [(lang$indication ?l) (lang$reference-arity ?l)])))

(24) (SET.FTN$function variable)
     (= (SET.FTN$source variable) lang$language)
     (= (SET.FTN$target variable) set.ftn$function)
     (= (SET.FTN$composition [variable set.ftn$source]) elementary)
     (= (SET.FTN$composition [variable set.ftn$target]) lang$variable)
     (forall (?l (lang$language ?l))
         (= (set.ftn$composition [(element ?l) (variable ?l)]) (lang$projection ?l)))

(25) (SET.FTN$function function)
     (= (SET.FTN$source function) lang$language)
     (= (SET.FTN$target function) set.ftn$function)
     (= (SET.FTN$composition [function set.ftn$source]) composite)
     (= (SET.FTN$composition [function set.ftn$target]) lang$function)
     (forall (?l (lang$language ?l))
         (= (set.ftn$composition [(substitution ?l) (function ?l)])
            (rel$first (substitutable ?l))))

(26) (SET.FTN$function tuple)
     (= (SET.FTN$source tuple) lang$language)
     (= (SET.FTN$target tuple) set.ftn$function)
     (= (SET.FTN$composition [tuple set.ftn$source]) composite)
     (= (SET.FTN$composition [tuple set.ftn$target]) lang.tpl$tuple)
     (forall (?l (lang$language ?l))
         (= (set.ftn$composition [(substitution ?l) (tuple ?l)])
            (rel$second (substitutable ?l))))

(27) (SET.FTN$function resolution)
     (= (SET.FTN$source resolution) lang$language)
     (= (SET.FTN$target resolution) set.ftn$function)
     (= (SET.FTN$composition [resolution set.ftn$source]) composite)
     (= (SET.FTN$composition [resolution set.ftn$target])
        (SET.FTN$composition [substitutable rel$extent]))
     (forall (?l (lang$language ?l))
         (and (= (set.ftn$composition [(resolution ?l) (substitution ?l)])
                 (set.ftn$inclusion [(composite ?l) (term ?l)]))
              (forall (?f ((lang$function ?l) ?f) ?a ((lang.tpl$tuple ?l) ?a)
                       ((substitutable ?l) [?f ?a]))
                  (= ((resolution ?l) ((substitution ?l) [?f ?a])) [?f ?a])))))
```

Term Tuples

`lang.tpl`

- We next give a pointwise recursive definition for the set of tuples of $\boldsymbol{L}$-terms $\mathit{term}^*(\boldsymbol{L})$. A term tuple $\varphi \in \mathit{term}^*(\boldsymbol{L})$ will have an index $\mathit{index}(\boldsymbol{L})(\varphi)$ and an arity $\mathit{arity}(\boldsymbol{L})(\varphi)$. Hence, there is
  - an *index* function $\S_L = \mathit{index}(\boldsymbol{L}) : \mathit{tpl}(\boldsymbol{L}) \to \wp\, \mathit{var}(\boldsymbol{L})$, and
  - an *arity* function $\#_L = \mathit{arity}(\boldsymbol{L}) : \mathit{tpl}(\boldsymbol{L}) \to \wp\, \mathit{var}(\boldsymbol{L})$.

  We use the notation

  $\varphi : B \multimap A$

  to denote a term with $\mathit{index}(\boldsymbol{L})(\varphi) = B$ and $\mathit{arity}(\boldsymbol{L})(\varphi) = A$. As we shall see below, term tuples will form the morphisms in a Kleisli-like category that will be used as the source category for the algebras-as-functors. Signatures could be used in place of arities for the objects of this category. However, arities are closer to the traditional objects used in functorial semantics.

```
(1) (SET.FTN$function tuple)
    (= (SET.FTN$source tuple) lang$language)
    (= (SET.FTN$target tuple) set$set)
    (forall (?l (lang$language ?l))
        (<=> ((tuple ?l) ?t)
             (and (set.ftn$function ?t)
                  (= (set.ftn$source ?t) ((index ?l) ?t))
                  (= (set.ftn$target ?t) (term ?l))
                  (forall (?x (((index ?l) ?t) ?x))
                      (and (= ((lang.term$arity ?l)(?t ?x)) ((arity ?l) ?t))
                           (= ((lang.term$type ?l) (?t ?x))
                              ((lang$reference ?l) ?x)))))))

(2) (SET.FTN$function index)
    (= (SET.FTN$source index) language)
    (= (SET.FTN$target index) set.ftn$function)
    (= (SET.FTN$composition [index set.ftn$source]) tuple)
    (= (SET.FTN$composition [index set.ftn$target])
       (SET.FTN$composition [lang$variable set$power]))

(3) (SET.FTN$function arity)
    (= (SET.FTN$source arity) language)
    (= (SET.FTN$target arity) set.ftn$function)
    (= (SET.FTN$composition [arity set.ftn$source]) tuple)
    (= (SET.FTN$composition [arity set.ftn$target])
       (SET.FTN$composition [lang$variable set$power]))
```

Constructors

- For any indicia (subset of variables) $A \subseteq \mathit{var}(\boldsymbol{L})$ there is an *empty* or *counique* term tuple

  $0_A : \varnothing \rightharpoonup A$

  with $\mathit{index}(\boldsymbol{L})(0_A) = \varnothing$ and $\mathit{arity}(\boldsymbol{L})(0_A) = A$. Hence, there is

  – an *empty* function $\mathit{empty}(\boldsymbol{L}) : \wp \mathit{var}(\boldsymbol{L}) \to \mathit{tpl}(\boldsymbol{L})$.

- Term tuples can be tupled. Two term tuples $\varphi_0, \varphi_1 \in \mathit{term}^*(\boldsymbol{L})$ are *tuplable* when they share a common arity $\mathit{arity}(\boldsymbol{L})(\varphi_0) = \mathit{arity}(\boldsymbol{L})(\varphi_1) = A$. For any two tuplable term tuples

  $\varphi_0 : A_0 \rightharpoonup A$ and $\varphi_1 : A_1 \rightharpoonup A$

  there is a binary term *tupling*

  $[\varphi_0, \varphi_1] : A_0 + A_1 \rightharpoonup A$

  with $\mathit{index}(\boldsymbol{L})([\varphi_0, \varphi_1]) = \mathit{index}(\boldsymbol{L})(\varphi_0) + \mathit{index}(\boldsymbol{L})(\varphi_1) = A_0 + A_1$ and $\mathit{arity}(\boldsymbol{L})([\varphi_0, \varphi_1]) = \mathit{arity}(\boldsymbol{L})(\varphi_0) = A$. The sum symbol here denotes the coproduct in the Kleisli-like category of term tuples. Strictly speaking we should use the terminology "copairing" instead of "tupling." Tupling is defined as the comediator of the binary coproduct cocone associated with a tuplable pair. Hence, there is a

  – a *tupling* function $[\text{-}, \text{-}]_{\boldsymbol{L}} : \mathit{tuplable}(\boldsymbol{L}) \to \mathit{tpl}(\boldsymbol{L})$.

  Tupling is associative: $[\varphi_0, [\varphi_1, \varphi_2]] = [[\varphi_0, \varphi_1], \varphi_2]$. Tupling, on either side, with the empty tuple is the identity: $[\varphi, 0_A] = \varphi$ and $[0_A, \varphi] = \varphi$.

- Terms are tuples. An indexed term (x, t) consisting of a variable $x \in \mathit{var}(\boldsymbol{L})$ and a term

  $t : A \rhd \alpha$

  with $\mathit{refer}(\boldsymbol{L})(x) = \alpha = \mathit{typ}(\boldsymbol{L})(t)$ becomes a *singleton* term tuple or *atom*

  $\{(x, t)\} : \{x\} \rightharpoonup A$

  with $\mathit{index}(\boldsymbol{L})(\{(x, t)\}) = \{x\}$ and $\mathit{arity}(\boldsymbol{L})(\{(x, t)\}) = \mathit{arity}(\boldsymbol{L})(t) = A$. Hence, there is a

  – a *singleton* function $\{\text{-}\}_{\boldsymbol{L}} : \mathit{ext}(\mathit{ind\text{-}term}(\boldsymbol{L})) \to \mathit{tpl}(\boldsymbol{L})$.

- In summary, for any term tuple $\varphi : B \rightharpoonup A$

  – $B = \varnothing$ iff then $\varphi = 0_A$;

  – $B = \{x\}$ iff $\varphi = \{(x, t)\}$ for a unique indexed term (x, t); and

  – $B = A_0 + A_1$ iff $\varphi = [\varphi_0, \varphi_1]$ for a tuplable pair $\varphi_0 : A_0 \rightharpoonup A$ and $\varphi_1 : A_1 \rightharpoonup A$.

```
(4) (SET.FTN$function empty)
    (= (SET.FTN$source empty) language)
    (= (SET.FTN$target empty) set.ftn$function)
    (= (SET.FTN$composition [empty set.ftn$source])
       (SET.FTN$composition [lang$variable set$power]))
    (= (SET.FTN$composition [empty set.ftn$target]) tuple)
    (forall (?l (lang$language ?l))
        (and (= (set.ftn$composition [(empty ?l) (index ?l)])
                ((set.ftn$constant
                    [(set$power (lang$variable ?l)) (set$power (lang$variable ?l))])
                     set.col$initial))
             (= (set.ftn$composition [(empty ?l) (arity ?l)])
                (set.ftn$identity (set$power (lang$variable ?l))))
             (forall (?a (set$subset ?a (lang$variable ?l)))
                 (= ((empty ?l) ?a) (set.col$counique (term ?l)))))))

(5) (SET.FTN$function tuplable-opspan)
    (= (SET.FTN$source tuplable-opspan) lang$language)
    (= (SET.FTN$target tuplable-opspan) set.lim.pbk$opspan)
    (= (SET.FTN$composition [tuplable-opspan set.lim.pbk$set1]) tuple)
    (= (SET.FTN$composition [tuplable-opspan set.lim.pbk$set2]) tuple)
    (= (SET.FTN$composition [tuplable-opspan set.lim.pbk$opvertex])
       (SET.FTN$composition [lang$variable set$power]))
    (= (SET.FTN$composition [tuplable-opspan set.lim.pbk$opfirst]) lang$arity)
    (= (SET.FTN$composition [tuplable-opspan set.lim.pbk$opsecond]) lang$arity)
```

```
(6) (SET.FTN$function tuplable)
    (= (SET.FTN$source tuplable) lang$language)
    (= (SET.FTN$target tuplable) rel$relation)
    (= tuplable (SET.FTN$composition [tuplable-opspan set.lim.pbk$relation]))

(7) (SET.FTN$function tupling-cocone)
    (= (SET.FTN$source tupling-cocone) lang$language)
    (= (SET.FTN$target tupling-cocone) set.ftn$function)
    (= (SET.FTN$composition [tupling-cocone set.ftn$source])
       (SET.FTN$composition [tuplable rel$extent]))
    (= (SET.FTN$composition [tupling-cocone set.ftn$target]) lang.tpl.colim.coprod2$cocone)
    (forall (?l (lang$language ?l))
        (and (= (set.ftn$composition [(tupling-cocone ?l) (lang.tpl.colim.coprod2$opfirst ?l)])
                (rel$first (tuplable ?l)))
             (= (set.ftn$composition [(tupling-cocone ?l) (lang.tpl.colim.coprod2$opsecond ?l)])
                (rel$second (tuplable ?l)))))

(8) (SET.FTN$function tupling)
    (= (SET.FTN$source tupling) lang$language)
    (= (SET.FTN$target tupling) set.ftn$function)
    (= (SET.FTN$composition [tupling set.ftn$source])
       (SET.FTN$composition [tuplable rel$extent]))
    (= (SET.FTN$composition [tupling set.ftn$target]) tuple)
    (forall (?l (lang$language ?l))
        (= (tupling ?l)
           (set.ftn$composition [(tupling-cocone ?l) (lang.tpl.colim.coprd2$comediator ?l)]))))

(9) (SET.FTN$function singleton)
    (SET.FTN$function atom)
    (= atom singleton)
    (= (SET.FTN$source singleton) language)
    (= (SET.FTN$target singleton) set.ftn$function)
    (= (SET.FTN$composition [singleton set.ftn$source])
       (SET.FTN$composition [lang.term$indexed-term rel$extent]))
    (= (SET.FTN$composition [singleton set.ftn$target]) tuple)
    (forall (?l (lang$language ?l))
        (and (= (set.ftn$composition [(singleton ?l) (index ?l)])
                (set.ftn$composition
                    [(rel$first (lang.term$indexed-term ?l)) (set$singleton (lang$variable ?l))]))
             (= (set.ftn$composition [(singleton ?l) (arity ?l)])
                (set.ftn$composition
                    [(rel$second (lang.term$indexed-term ?l)) (lang.term$arity ?l)])))))

(10) (forall (?l (lang$language ?l)
              ?t0 ((tuple ?l) ?t0) ?t1 ((tuple ?l) ?t1) ?t2 ((tuple ?l) ?t2)
              ((tuplable ?l) ?t0 ?t1) ((tuplable ?l) ?t1 ?t2))
         (= ((tupling ?l) [?t0 ((tupling ?l) [?t1 ?t2])])
            ((tupling ?l) [((tupling ?l) [?t0 ?t1]) ?t2]))))

(11) (forall (?l (lang$language ?l) ?t ((tuple ?l) ?t))
         (and (= ((tupling ?l) [?t ((empty ?l) ((arity ?l) ?t))]) ?t)
              (= ((tupling ?l) [((empty ?l) ((arity ?l) ?t)) ?t]) ?t)))

(12) (forall (?l (lang$language ?l) ?t ((tuple ?l) ?a))
         (and (<=> (= ((index ?l) ?a) set.lim$initial)
                   (= ?a ((empty ?l) ((arity ?l) ?a))))
              (<=> (exists (?x ((lang$variable ?l) ?x))
                       (= ((index ?l) ?a) ((set$singleton (lang$variable ?l)) ?x)))
                   (exists (?t ((lang.term$term ?l) ?t) (lang.term$indexed-term ?x ?t))
                       (= ?a ((singleton ?l) [?x ?t]))))
              (<=> (exists (?x0 (set$subset ?x0 (lang$variable ?l))
                            ?x1 (set$subset ?x1 (lang$variable ?l))
                            (set$disjoint ?X ?Y))
                       (= ((index ?l) ?a) (set$binary-union [?x0 ?x1])))
                   (exists (?a0 ((tuple ?l) ?a0) ?a1 ((tuple ?l) ?a1) ((tuplable ?l) ?a0 ?a1))
                       (and (= ?a ((tupling ?l) [?a0 ?a1])
                                  (= ((index ?l) ?a0) ?x0) (= ((index ?l) ?a1) ?x1))))))))
```

Booleans

A term tuple is empty when it is built by the empty constructor. A term tuple is not empty when it is not so built. A term is an atom when it is built by the singleton constructor. These Boolean tests are used to define composite constructors and the domain of selectors. Obviously, the empty and non-empty term tuples comprise a partition for the set of term tuples.

```
(13) (SET.FTN$function is-empty)
     (= (SET.FTN$source is-empty) language)
     (= (SET.FTN$target is-empty) set$set)
     (forall (?l (lang$language ?l))
         (and (set$subset (is-empty ?l) (tuple ?l))
              (= (is-empty ?l) (set.ftn$image (empty ?l)))))

(14) (SET.FTN$function is-nonempty)
     (= (SET.FTN$source is-nonempty) language)
     (= (SET.FTN$target is-nonempty) set$set)
     (forall (?l (lang$language ?l))
         (and (set$subset (is-nonempty ?l) (tuple ?l))
              (= (is-nonempty ?l) (set$difference [(tuple ?l) (is-empty ?l)])))))

(15) (SET.FTN$function is-singleton)
     (SET.FTN$function is-atom)
     (= is-atom is-singleton)
     (= (SET.FTN$source is-singleton) language)
     (= (SET.FTN$target is-singleton) set$set)
     (forall (?l (lang$language ?l))
         (and (set$subset (is-singleton ?l) (tuple ?l))
               (= (is-singleton ?l) (set.ftn$image (singleton ?l)))))
```

Composite Constructors

- Terms can be inserted breadthwise into term tuples. An indexed term $(x, t) \in \mathit{term}(\boldsymbol{L})$ and a term tuple $\varphi \in \mathit{term}^*(\boldsymbol{L})$ are an *insertible* pair when the arity of the term t is the same as the arity of the term tuple φ: $\mathit{arity}(\boldsymbol{L})(t) = \mathit{arity}(\boldsymbol{L})(\varphi)$. The pair (x, t) and φ is insertible iff the pair $\{(x, t)\}$ and φ is tuplable. For any insertible pair

  $t : A \rhd \alpha$ and $\varphi : B \rightharpoonup A$ and $x \notin B$

  there is an *insertion* term tuple defined as the tupling of the singleton

  $\langle (x, t), \varphi \rangle = [\{(x, t)\}, \varphi] : (\{x\}+B) \rightharpoonup A$

  with $\mathit{index}(\boldsymbol{L})(\langle (x, t), \varphi \rangle) = \{x\}+\mathit{index}(\boldsymbol{L})(\varphi) = \{x\}+B$ and $\mathit{arity}(\boldsymbol{L})(\langle (x, t), \varphi \rangle) = \mathit{arity}(\boldsymbol{L})(\varphi) = A$.

  Hence, there is

  – an *insertion* function $\mathit{insert}(\boldsymbol{L}) : \mathit{ext}(\mathit{insertible}(\boldsymbol{L})) \to \mathit{tpl}(\boldsymbol{L})$.

  Insertion into the empty tuple reduces the insertion function to the singleton function

  $\{(x, t)\} = \langle (x, t), 0_A \rangle$.

- For any two subsets of variables $B \subseteq A$ ordered by inclusion, there is an inclusion term tuple

  $\mathit{incl}_{B, A} : B \rightharpoonup A$

  with $\mathit{index}(\boldsymbol{L})(\mathit{incl}_{B, A}) = B$ and $\mathit{arity}(\boldsymbol{L})(\mathit{incl}_{B, A}) = A$.

- Term tuples can be renamed. A *renamable pair* (n, φ) consists of a renaming n and a term tuple φ where the codomain of the renaming is the index of the term tuple $\mathit{cod}(\boldsymbol{L})(n) = \mathit{index}(\boldsymbol{L})(\varphi)$. For any renamable pair

  $n : C \leftrightarrow B$ and $\varphi : B \rightharpoonup A$

  there is a *renamed* term tuple defined as the function composition

  $n \circ \varphi = n \cdot \varphi : C \rightharpoonup A$

  with $\mathit{index}(\boldsymbol{L})(n \circ \varphi) = \mathit{dom}(\boldsymbol{L})(n) = C$ and $\mathit{arity}(\boldsymbol{L})(n \circ \varphi) = \mathit{arity}(\boldsymbol{L})(\varphi) = A$. Hence, there is

  – a *renaming* function $\mathit{rename}(\boldsymbol{L}) : \mathit{ext}(\mathit{renamable}(\boldsymbol{L})) \to \mathit{tpl}(\boldsymbol{L})$.

```
(16) (SET.FTN$function insertible-opspan)
     (= (SET.FTN$source insertible-opspan) lang$language)
     (= (SET.FTN$target insertible-opspan) set.lim.pbk$opspan)
     (= (SET.FTN$composition [insertible-opspan set.lim.pbk$set1]) lang.term$indexed-term)
     (= (SET.FTN$composition [insertible-opspan set.lim.pbk$set2]) tuple)
     (= (SET.FTN$composition [insertible-opspan set.lim.pbk$opvertex])
        (SET.FTN$composition [lang$variable set$power]))
     (forall (?l (lang$language ?l))
         (and (= (set.lim.pbk$opfirst (insertible-opspan ?l))
                 (set.ftn$composition
                     [(rel$second (lang.term$indexed-term ?l)) (lang.term$arity ?l)]))
              (= (set.lim.pbk$opsecond (insertible-opspan ?l))
                 (arity ?l))))

(17) (SET.FTN$function insertible)
     (= (SET.FTN$source insertible) lang$language)
     (= (SET.FTN$target insertible) rel$relation)
     (= insertible (SET.FTN$composition [insertible-opspan set.lim.pbk$relation]))

(18) (SET.FTN$function insertion)
     (= (SET.FTN$source insertion) lang$language)
     (= (SET.FTN$target insertion) set.ftn$function)
     (= (SET.FTN$composition [insertion set.ftn$source])
        (SET.FTN$composition [insertible rel$extent]))
     (= (SET.FTN$composition [insertion set.ftn$target]) tuple)
     (forall (?l (lang$language ?l)
              ?a ((lang.term$indexed-term ?l) ?a) ?b ((tuple ?l) ?b)
              (insertible ?a ?b))
         (= ((insertion ?l) [?a ?b])
            ((tupling ?l) [((singleton ?l) ?a) ?b])))
```

```
(19) (SET.FTN$function inclusion)
     (= (SET.FTN$source inclusion) lang$language)
     (= (SET.FTN$target inclusion) set.ftn$function)
     (= (SET.FTN$composition [inclusion set.ftn$source])
        (SET.FTN$composition [lang$variable (SET.FTN$composition [ord$subset ord$extent])]))
     (= (SET.FTN$composition [inclusion set.ftn$target]) tuple)
     (forall (?l (lang$language ?l))
         (and (= (set.ftn$composition [(inclusion ?l) (index ?l)])
                 (ord$first (ord$subset (lang$variable ?l))))
              (= (set.ftn$composition [(inclusion ?l) (arity ?l)])
                 (ord$second (ord$subset (lang$variable ?l))))
              (forall (?b ?a (set$subset ?b ?a)
                       ?x (lang$variable ?l) ?x) (?b ?x))
                  (= (((inclusion ?l) [?b ?a]) ?x)
                     ((lang$element ?l) [?a ?x])))))

(20) (SET.FTN$function renamable-opspan)
     (= (SET.FTN$source renamable-opspan) lang$language)
     (= (SET.FTN$target renamable-opspan) set.lim.pbk$opspan)
     (= (SET.FTN$composition [renamable-opspan set.lim.pbk$set1]) lang$renaming)
     (= (SET.FTN$composition [renamable-opspan set.lim.pbk$set2]) tuple)
     (= (SET.FTN$composition [renamable-opspan set.lim.pbk$opvertex])
        (SET.FTN$composition [lang$variable set$power]))
     (= (SET.FTN$composition [renamable-opspan set.lim.pbk$opfirst]) lang$codomain)
     (= (SET.FTN$composition [renamable-opspan set.lim.pbk$opsecond]) index)

(21) (SET.FTN$function renamable)
     (= (SET.FTN$source renamable) lang$language)
     (= (SET.FTN$target renamable) rel$relation)
     (= renamable (SET.FTN$composition [renamable-opspan set.lim.pbk$relation]))

(22) (SET.FTN$function renaming)
     (= (SET.FTN$source renaming) lang$language)
     (= (SET.FTN$target renaming) set.ftn$function)
     (= (SET.FTN$composition [renaming set.ftn$source])
        (SET.FTN$composition [renamable rel$extent]))
     (= (SET.FTN$composition [renaming set.ftn$target]) tuple)
     (forall (?l (lang$language ?l))
         (and (= (set.ftn$composition [(renaming ?l) (index ?l)])
                 (set.ftn$composition [(rel$first (renamable ?l)) (lang$domain ?l)])))
              (= (set.ftn$composition [(renaming ?l) (arity ?l)])
                 (set.ftn$composition [(rel$second (renamable ?l)) (arity ?l)]))
              (forall (?n ((lang$renaming ?l) ?n) ?a ((tuple ?l) ?a) (renamable ?n ?a))
                  (= ((renaming ?l) [?n ?a])
                     (set.ftn$composition [?n ?a])))))
```

Selectors

- The tuple index function

  $index(\boldsymbol{L}) : tpl(\boldsymbol{L}) \to \wp var(\boldsymbol{L})$

  can serve as a colimit arity.

```
(23) (SET.FTN$function index-arity)
     (= (SET.FTN$source index-arity) set$set)
     (= (SET.FTN$target index-arity) set.col.art$arity)
     (= (SET.FTN$composition [index-arity set.col.art$index]) tuple)
     (= (SET.FTN$composition [index-arity set.col.art$base]) lang$variable)
     (= (SET.FTN$composition [index-arity set.col.art$function]) index)
```

- Any type language $\boldsymbol{L}$ has a set of tuple cases

  $case(\boldsymbol{L}) = \sum index\text{-}arity(\boldsymbol{L})$

  $= \sum_{\varphi \in tpl(\boldsymbol{L})} index\text{-}arity(\boldsymbol{L})(\varphi)$

  $= \{(\varphi, x) \mid \varphi \in tpl(\boldsymbol{L}), x \in index(\boldsymbol{L})(\varphi)\},$

  that is the coproduct of its index arity.

  For any type language $\boldsymbol{L}$ and any term tuple $\varphi \in tpl(\boldsymbol{L})$ there is a case *injection* function:

  $inj(\boldsymbol{L})(\varphi) : index(\boldsymbol{L})(\varphi) \to case(\boldsymbol{L})$

  defined by $inj(\boldsymbol{L})(\varphi)(x) = (\varphi, x)$ for all variables $x \in index(\boldsymbol{L})(\varphi)$. Obviously, the injections are injective. They commute (Diagram 2) with projection and inclusion.

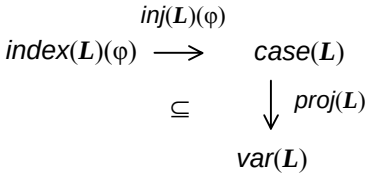

Diagram 2: Coproduct

```
(24) (SET.FTN$function case)
     (= (SET.FTN$source case) lang$language)
     (= (SET.FTN$target case) set$set)
     (= case (SET.FTN$composition [index-arity set.col.art$colimit]))

(25) (KIF$function injection)
     (= (KIF$source injection) lang$language)
     (= (KIF$target injection) SET.FTN$function)
     (= injection (SET.FTN$composition [index-arity set.col.art$injection]))
```

- Any type language $\boldsymbol{L}$ defines *indication* and *projection* functions based on its index arity (Figure 4):

  $indic(\boldsymbol{L}) : case(\boldsymbol{L}) \to tpl(\boldsymbol{L}),$

  $proj(\boldsymbol{L}) : case(\boldsymbol{L}) \to var(\boldsymbol{L}).$

  These are defined by

  $indic(\boldsymbol{L})((\varphi, x)) = \varphi$ and $proj(\boldsymbol{L})((\varphi, x)) = x$

  for all term tuples $\varphi \in tpl(\boldsymbol{L})$ and all variables $x \in index(\boldsymbol{L})(\varphi)$.

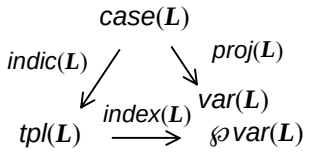

Figure 4: Indication and projection

```
(26) (SET.FTN$function indication)
     (= (SET.FTN$source indication) lang$language)
     (= (SET.FTN$target indication) set.ftn$function)
     (= (SET.FTN$composition [indication set.ftn$source]) case)
     (= (SET.FTN$composition [indication set.ftn$target]) tuple)
     (= indication (SET.FTN$composition [index-arity set.col.art$indication]))

(27) (SET.FTN$function projection)
     (= (SET.FTN$source projection) lang$language)
     (= (SET.FTN$target projection) set.ftn$function)
     (= (SET.FTN$composition [projection set.ftn$source]) case)
     (= (SET.FTN$composition [projection set.ftn$target]) lang$variable)
     (= projection (SET.FTN$composition [index-arity set.col.art$projection]))
```

- Terms can be selected out of a term tuple, leaving a remainder term tuple. For any case (φ, x) consisting of a term tuple $\varphi \in tpl(\boldsymbol{L})$ and a variable $x \in index(\boldsymbol{L})(\varphi)$

  $x \in B$ and $\varphi : B \rightharpoonup A$

there is a *selection* term

$\sigma_x(\varphi) : A \rhd x$

and a *remainder* term tuple

$\rho_x(\varphi) : (B - \{x\}) \rightharpoonup A$.

The source of selection and remainder is the set of cases. Hence, there is

– an *selection* function $\mathit{select}(\boldsymbol{L}) : \mathit{case}(\boldsymbol{L}) \to \mathit{term}(\boldsymbol{L})$, and

– a *remainder* function $\mathit{rest}(\boldsymbol{L}) : \mathit{case}(\boldsymbol{L}) \to \mathit{tpl}(\boldsymbol{L})$.

Clearly, the insertion of a term indexed by variable x followed by a selection of the x^{th} component term is the identity on terms.

```
(28) (SET.FTN$function selection)
     (= (SET.FTN$source selection) lang$language)
     (= (SET.FTN$target selection) set.ftn$function)
     (= (SET.FTN$composition [selection set.ftn$source]) case)
     (= (SET.FTN$composition [selection set.ftn$target]) lang.term$term)
     (forall (?l (lang$language ?l))
         (and (= (set.ftn$composition [(selection ?l) (lang.term$type ?l)])
                 (set.ftn$composition [(projection ?l) (lang$reference ?l)]))
              (= (set.ftn$composition [(selection ?l) (lang.term$arity ?l)])
                 (set.ftn$composition [(indication ?l) (arity ?l)])))
              (forall (?a ((tuple ?l) ?a) ?x ((lang$variable ?l) ?x) ((case ?l) [?a ?x])
                  (= ((selection ?l) [?a ?x]) (?a ?x)))))

(29) (SET.FTN$function remainder)
     (SET.FTN$function rest)
     (= rest remainder)
     (= (SET.FTN$source remainder) lang$language)
     (= (SET.FTN$target remainder) set.ftn$function)
     (= (SET.FTN$composition [remainder set.ftn$source]) case)
     (= (SET.FTN$composition [remainder set.ftn$target]) tuple)
     (forall (?l (lang$language ?l))
         (and (= (set.ftn$composition [(remainder ?l) (arity ?l)])
                 (set.ftn$composition [(indication ?l) (arity ?l)]))
              (forall (?a ((tuple ?l) ?a) ?x ((lang$variable ?l) ?x) ((case ?l) [?a ?x]))
                  (and (= ((index ?l) ((remainder ?l) [?a ?b]))
                          (set$difference
                              [((arity ?l) ?a) ((set.ftn$singleton (lang$variable ?l)) ?x)]))
                       (forall (?y (((index ?l) ?a) ?y) (not (= ?y ?x)))
                           (= (((remainder ?l) [?a ?x]) ?y) (?a ?y)))))))

(30) (forall (?l (lang$language ?l)
              ?t ((lang.term$term ?l) ?t)
              ?x ((lang$variable ?l) ?x)
              ((lang.term$indexed-term ?l) [?x ?t])
              ?b ((tuple ?l) ?b)
              ((insertible ?l) [?x ?t] ?b))
         (and (= ((selection ?l) [((insertion ?l) [?x ?t] ?b) ?x]) ?t)
              (= ((remainder ?l) [((insertion ?l) [?x ?t] ?b) ?x]) ?b)))
```

Category-theoretic Operations

○ Term tuples can be composed. Two term tuples $\psi, \varphi \in \mathit{term}^*(\boldsymbol{L})$ are *composable* when the arity of the first is the index of the second $\mathit{arity}(\boldsymbol{L})(\psi) = \mathit{index}(\boldsymbol{L})(\varphi)$. For any two composable term tuples

$\psi : C \rightharpoonup B$ and $\varphi : B \rightharpoonup A$

there is a *composition* term tuple

$\psi \circ \varphi : C \rightharpoonup A$

with $\mathit{index}(\boldsymbol{L})(\psi \circ \varphi) = \mathit{index}(\boldsymbol{L})(\psi) = C$ and $\mathit{arity}(\boldsymbol{L})(\psi \circ \varphi) = \mathit{arity}(\boldsymbol{L})(\varphi) = A$. The axiomatic definition proceeds by selection then term substitution:

$(\psi \circ \varphi)(z) = \sigma_z(\psi)[\varphi] : A \rhd \alpha$

for every variable $z \in C = \mathit{index}(\boldsymbol{L})(\psi)$ with $\mathit{refer}(\boldsymbol{L})(z) = \alpha$. Hence, there is a

– a *composition* function $\circ_{\boldsymbol{L}} = \mathit{comp}(\boldsymbol{L}) : \mathit{composable}(\boldsymbol{L}) \to \mathit{tpl}(\boldsymbol{L})$.

```
(31) (SET.FTN$function composable-opspan)
     (= (SET.FTN$source composable-opspan) lang$language)
     (= (SET.FTN$target composable-opspan) set.lim.pbk$opspan)
     (= (SET.FTN$composition [composable-opspan set.lim.pbk$set1]) tuple)
     (= (SET.FTN$composition [composable-opspan set.lim.pbk$set2]) tuple)
     (= (SET.FTN$composition [composable-opspan set.lim.pbk$opvertex])
        (SET.FTN$composition [lang$variable set$power]))
     (= (SET.FTN$composition [composable-opspan set.lim.pbk$opfirst]) arity)
     (= (SET.FTN$composition [composable-opspan set.lim.pbk$opsecond]) index)

(32) (SET.FTN$function composable)
     (= (SET.FTN$source composable) lang$language)
     (= (SET.FTN$target composable) rel$relation)
     (= composable (SET.FTN$composition [composable-opspan set.lim.pbk$relation]))

(33) (SET.FTN$function composition)
     (= (SET.FTN$source composition) lang$language)
     (= (SET.FTN$target composition) set.ftn$function)
     (= (SET.FTN$composition [composition set.ftn$source])
        (SET.FTN$composition [composable rel$extent]))
     (= (SET.FTN$composition [composition set.ftn$target]) tuple)
     (forall (?l (lang$language ?l)
              ?b ((tuple ?l) ?b) ?a ((tuple ?l) ?a)
              (composable ?b ?a))
         (and (= ((index ?l) ((composition ?l) [?b ?a])) ((index ?l) ?b))
              (= ((arity ?l) ((composition ?l) [?b ?a])) ((arity ?l) ?a))
              (forall (?z (((index ?l) ?b) ?z))
                  (= (((composition ?l) [?b ?a]) ?z)
                     ((lang.term$term-substitution ?l) [((selection ?l) [?b ?z]) ?a]))))))
```

o Composition satisfies the usual *associative law*.

```
(forall (?l (lang$language ?l)
         ?a1 ((tuple ?l) ?a1) ?a2 ((tuple ?l) ?a2) ?a3 ((tuple ?l) ?a3))
         (composable ?a1 ?a2) (composable ?a2 ?a3))
    (= (composition [?a1 (composition [?a2 ?a3])])
       (composition [(composition [?a1 ?a2]) ?a3])))
```

○ For any subset of variables $A \subseteq \mathit{var}(\boldsymbol{L})$ regarded as an arity, there is an *identity* term tuple

$\mathit{id}_A : A \rightharpoonup A$

with $\mathit{index}(\boldsymbol{L})(\mathit{id}_A) = A$ and $\mathit{arity}(\boldsymbol{L})(\mathit{id}_A) = A$. Hence, there is a

– an *identity* function $\mathit{id}(\boldsymbol{L}) : \wp \mathit{var}(\boldsymbol{L}) \to \mathit{tpl}(\boldsymbol{L})$.

```
(34) (SET.FTN$function identity)
     (= (SET.FTN$source identity) lang$language)
     (= (SET.FTN$target identity) set.ftn$function)
     (= (SET.FTN$composition [identity set.ftn$source])
        (SET.FTN$composition [lang$variable set$power]))
     (= (SET.FTN$composition [identity set.ftn$target]) tuple)
     (forall (?l (lang$language ?l))
```

```
        (and (= (set.ftn$composition [(identity ?l) (index ?l)])
                (set.ftn$identity (set$power (lang$variable ?l)))
             (= (set.ftn$composition [(identity ?l) (arity ?l)])
                (set.ftn$identity (set$power (lang$variable ?l)))
             (forall (?X (set$subset ?X (lang$variable ?l)) ?x (?X ?x))
                 (= (((identity ?l) ?X) ?x) (lang.term$element [?X ?x])))))
```

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

```
    (forall (?l (lang$language ?l))
            ?a ((tuple ?l) ?a))
        (and (= (composition [(identity (index ?a)) ?a]) ?a)
             (= (composition [?a (identity (arity ?a))]) ?a)))
```

Colimits

`lang.tpl.col`

Here we present axioms for coproducts in the category of term tuples for any type language $\boldsymbol{L}$. We assert the existence of initial objects and binary coproducts and describe their nature.

The Initial Object

- The empty subset of variables $\varnothing$ serves as the initial object in the category of term tuples.

```
(1) (SET.FTN$function initial)
    (= (SET.FTN$source initial) lang$language)
    (= (SET.FTN$target initial) set$set)
    (forall (?l (lang$language ?l))
        (= (initial ?l) set.col$initial))
```

- For any subset of variables $A \subseteq \mathit{var}(\boldsymbol{L})$, regarded as an arity, there is an *counique* term tuple

  $0_A : \varnothing \rightharpoonup A$

  with $\mathit{index}(\boldsymbol{L})(0_A) = \varnothing$ and $\mathit{arity}(\boldsymbol{L})(0_A) = A$.

```
(2) (SET.FTN$function counique)
    (= (SET.FTN$source counique) lang$language)
    (= (SET.FTN$target counique) set.ftn$function)
    (= (SET.FTN$composition [counique set.ftn$source])
       (SET.FTN$composition [lang$variable set$power]))
    (= (SET.FTN$composition [counique set.ftn$target]) lang.tpl$tuple)
    (forall (?l (lang$language ?l))
        (and (= (set.ftn$composition [(counique ?l) (lang.tpl$index ?l)])
                ((set.ftn$constant
                    [(set$power (lang$variable ?l))
                     (set$power (lang$variable ?l))])
                 (initial ?l)))
             (= (set.ftn$composition [(counique ?l) (lang.tpl$arity ?l)])
                (set.ftn$identity (set$power (lang$variable ?l))))
             (= (counique ?l) (lang.tpl$empty ?l))))
```

Binary Coproducts

`lang.tpl.col.coprd2`

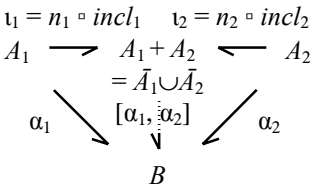

Diagram 1: Binary Coproduct

A *binary coproduct* of term tuples (Diagram 1) is a finite colimit for a diagram of shape *two* = • •. Such a diagram (of variable subsets and terms) is called a *pair* of subsets.

- A *pair* (of variable subsets) is the appropriate base diagram for a binary coproduct of term tuples. Each pair consists of two subsets of variables called *subset1* and *subset2*. We use either the generic term '`diagram`' or the specific term '`pair`' to denote the *pair* class. A pair is the special case of a general diagram of shape *two*.

```
(1) (SET.FTN$function diagram)
    (SET.FTN$function pair)
    (= (SET.FTN$source diagram) lang$language)
    (= (SET.FTN$target diagram) set$set)

(2) (SET.FTN$function subset1)
    (= (SET.FTN$source subset1) lang$language)
    (= (SET.FTN$target subset1) set.ftn$function)
    (= (SET.FTN$composition [subset1 set.ftn$source]) diagram)
    (= (SET.FTN$composition [subset1 set.ftn$target])
       (SET.FTN$composition [lang$variable set$power]))

(3) (SET.FTN$function subset2)
    (= (SET.FTN$source subset2) lang$language)
    (= (SET.FTN$target subset2) set.ftn$function)
    (= (SET.FTN$composition [subset2 set.ftn$source]) diagram)
    (= (SET.FTN$composition [subset2 set.ftn$target])
       (SET.FTN$composition [lang$variable set$power]))
```

- We want to axiomatize the existence of binary coproducts within the subsets of variables $\wp\,\mathit{var}(\boldsymbol{L})$ of a function type language $\boldsymbol{L}$. In order to realize this we assume the following rather significant property: for any pair of subsets $A_1, A_2 \subseteq \mathit{var}(\boldsymbol{L})$ there are renamings $n_1 : A_1 \leftrightarrow \bar{A}_1$ and $n_2 : A_2 \leftrightarrow \bar{A}_2$ such that $\bar{A}_1$ and $\bar{A}_2$ are disjoint $\bar{A}_1 \cap \bar{A}_2 = \varnothing$. Amongst other things this implies that a nonempty set of variables is infinite. Initially, axioms for a "coproduct family" of indicia (subsets of variable) were drawn up to be used in place of the full powerset of variables. But this has the negative effect of placing onerous restrictions on variables, in comparison with how variables are used in relation type languages. So we have abandoned this approach, and merely assume that the set of variables is constructed to satisfy this axiom. In particular, the special case $\mathit{var}(\boldsymbol{L}) = \mathit{entity}(\boldsymbol{L}) \times \mathsf{natno}$ obviously satisfies this property, even with infinite arities. As an extreme example of the latter, let A_1 and A_2 each be the complete set of variables, and define the renamings $n_1((\varepsilon, n)) = (\varepsilon, 2{*}n)$ and $n_2((\varepsilon, n)) = (\varepsilon, 2{*}n{+}1)$.

```
(4) (forall (?l (lang$language ?l)
             ?a1 (set$subset ?a1 (lang$variable ?l))
             ?a2 (set$subset ?a2 (lang$variable ?l)))
        (exists (?n1 ((lang$renaming ?l) ?n1)
                 ?n2 ((lang$renaming ?l) ?n2))
            (and (= ((domain ?l) ?n1) ?a1)
                 (= ((domain ?l) ?n2) ?a2)
                 (set$disjoint [((lang$codomain ?l) ?n1) ((lang$codomain ?l) ?n2)]))))
```

This property implies that the category of terms $\mathit{term}(\boldsymbol{A})$ has binary coproducts $A_0 + A_1 = \bar{A}_0 \cup \bar{A}_1$. Note that these are not canonical, and hence these coproducts are abstract not concrete. In particular, since there are many renamings possible, there are many coproducts for a particular pair of sets. But, as is well-known from category theory, all of these are isomorphic. To make this more concrete we will declare two choice functions that select two such renamings. Axioms 5–7 imply axiom 4.

```
(5) (SET.FTN$function renaming1)
    (= (SET.FTN$source renaming1) lang$language)
    (= (SET.FTN$target renaming1) set.ftn$function)
    (= (SET.FTN$composition [renaming1 set.ftn$source]) diagram)
    (= (SET.FTN$composition [renaming1 set.ftn$target]) lang$renaming)
```

```
    (forall (?l (lang$language ?l))
        (= (set.ftn$composition [(renaming1 ?l) (lang$domain ?l)]) (subset1 ?l)))

(6) (SET.FTN$function renaming2)
    (= (SET.FTN$source renaming2) lang$language)
    (= (SET.FTN$target renaming1) set.ftn$function)
    (= (SET.FTN$composition [renaming1 set.ftn$source]) diagram)
    (= (SET.FTN$composition [renaming1 set.ftn$target]) lang$renaming)
    (forall (?l (lang$language ?l))
        (= (set.ftn$composition [(renaming2 ?l) (lang$domain ?l)]) (subset2 ?l)))

(7) (forall (?l (lang$language ?l) ?p ((diagram ?l) ?p))
        (set$disjoint
            [((lang$codomain ?l) ((renaming1 ?l) ?p))
             ((lang$codomain ?l) ((renaming2 ?l) ?p))]))
```

- A *binary coproduct cocone* of term tuples is the appropriate cocone for a binary coproduct. A coproduct cocone consists of a pair of term tuples called *opfirst* and *opsecond*. These are required to have a common arity called the *opvertex* of the cocone.

```
(8) (SET.FTN$function cocone)
    (= (SET.FTN$source cocone) lang$language)
    (= (SET.FTN$target cocone) set$set)

(9) (SET.FTN$function cocone-diagram)
    (= (SET.FTN$source cocone-diagram) lang$language)
    (= (SET.FTN$target cocone-diagram) set.ftn$function)
    (= (SET.FTN$composition [cocone-diagram set.ftn$source]) cocone)
    (= (SET.FTN$composition [cocone-diagram set.ftn$target]) diagram)

(10) (SET.FTN$function opvertex)
     (= (SET.FTN$source opvertex) lang$language)
     (= (SET.FTN$target opvertex) set.ftn$function)
     (= (SET.FTN$composition [opvertex set.ftn$source]) cocone)
     (= (SET.FTN$composition [opvertex set.ftn$target])
        (SET.FTN$composition [lang$variable set$power]))

(11) (SET.FTN$function opfirst)
     (= (SET.FTN$source opfirst) lang$language)
     (= (SET.FTN$target opfirst) set.ftn$function)
     (= (SET.FTN$composition [opfirst set.ftn$source]) cocone)
     (= (SET.FTN$composition [opfirst set.ftn$target]) lang.tpl$tuple)
     (forall (?l (lang$language ?l))
         (and (= (set.ftn$composition [(opfirst ?l) (lang.tpl$index ?l)])
                 (set.ftn$composition [(cocone-diagram ?l) (subset1 ?l)]))
              (= (set.ftn$composition [(opfirst ?l) (lang.tpl$arity ?l)])
                 (opvertex ?l))))

(12) (SET.FTN$function opsecond)
     (= (SET.FTN$source opsecond) lang$language)
     (= (SET.FTN$target opsecond) set.ftn$function)
     (= (SET.FTN$composition [opsecond set.ftn$source]) cocone)
     (= (SET.FTN$composition [opsecond set.ftn$target]) lang.tpl$tuple)
     (forall (?l (lang$language ?l))
         (and (= (set.ftn$composition [(opsecond ?l) (lang.tpl$index ?l)])
                 (set.ftn$composition [(cocone-diagram ?l) (subset2 ?l)]))
              (= (set.ftn$composition [(opsecond ?l) (lang.tpl$arity ?l)])
                 (opvertex ?l))))
```

- The function '`(colimiting-cocone ?l)`' maps a pair of variable subsets to its binary coproduct (colimiting binary coproduct cocone). A colimiting binary coproduct cocone is the special case of a general colimiting cocone over a binary coproduct diagram (pair of variable subsets). The renamings of the inclusion term tuples

  $\iota_1 = n_1 \circ incl_1 : A_1 \rightharpoonup \bar{A}_1 \cup \bar{A}_2$ and $\iota_2 = n_2 \circ incl_2 : A_2 \rightharpoonup \bar{A}_1 \cup \bar{A}_2$

  serve as coproduct injections.

```
(13) (SET.FTN$function colimiting-cocone)
     (= (SET.FTN$source colimiting-cocone) lang$language)
     (= (SET.FTN$target colimiting-cocone) set.ftn$function)
```

```
        (= (SET.FTN$composition [colimiting-cocone set.ftn$source]) diagram)
        (= (SET.FTN$composition [colimiting-cocone set.ftn$source]) cocone)
        (forall (?l (lang$language ?l))
            (= (set.ftn$composition [(colimiting-cocone ?l) (cocone-diagram ?l)])
               (set.ftn$identity (diagram ?l))))

    (14) (SET.FTN$function colimit)
        (SET.FTN$function binary-coproduct)
        (= binary-coproduct colimit)
        (= (SET.FTN$source colimit) lang$language)
        (= (SET.FTN$source colimit) set.ftn$function)
        (= (SET.FTN$composition [colimit set.ftn$source]) diagram)
        (= (SET.FTN$composition [colimit set.ftn$source])
           (SET.FTN$composition [lang$variable set$power]))
        (forall (?l (lang$language ?l))
            (and (= (colimit ?l)
                    (set.ftn$composition [(colimiting-cocone ?l) (opvertex ?l)]))
                 (forall (?a1 ((set$power (lang$variable ?l)) ?a1)
                          ?a2 ((set$power (lang$variable ?l)) ?a2))
                     (= ((colimit ?l) [?a1 ?a2])
                        (set$binary-union
                            [((lang$codomain ?l) ((renaming1 ?l) [?a1 ?a2]))
                             ((lang$codomain ?l) ((renaming2 ?l) [?a1 ?a2]))])))))

    (15) (SET.FTN$function injection1)
        (= (SET.FTN$source injection1) lang$language)
        (= (SET.FTN$target injection1) set.ftn$function)
        (= (SET.FTN$composition [injection1 set.ftn$source]) diagram)
        (= (SET.FTN$composition [injection1 set.ftn$target]) lang.tpl$tuple)
        (forall (?l (lang$language ?l))
            (and (= (injection1 ?l)
                    (set.ftn$composition [(colimiting-cocone ?l) (opfirst ?l)]))
                 (forall (?p ((diagram ?l) ?p))
                     (= ((injection1 ?l) ?p)
                        (lang.tpl$renaming
                            [((renaming1 ?l) ?p)
                             ((lang.tpl$inclusion ?l)
                                 [((lang$codomain ?l) ((renaming1 ?l) ?p))
                                  ((colimit ?l) ?p)])])))))

    (16) (SET.FTN$function injection2)
        (= (SET.FTN$source injection2) lang$language)
        (= (SET.FTN$target injection2) set.ftn$function)
        (= (SET.FTN$composition [injection2 set.ftn$source]) diagram)
        (= (SET.FTN$composition [injection2 set.ftn$target]) lang.tpl$tuple)
        (forall (?l (lang$language ?l))
            (and (= (injection2 ?l)
                    (set.ftn$composition [(colimiting-cocone ?l) (opsecond ?l)]))
                 (forall (?p ((diagram ?l) ?p))
                     (= ((injection2 ?l) ?p)
                        (lang.tpl$renaming
                            [((renaming2 ?l) ?p)
                             ((lang.tpl$inclusion ?l)
                                 [((lang$codomain ?l) ((renaming2 ?l) ?p))
                                  ((colimit ?l) ?p)])])))))
```

- For any binary coproduct cocone, there is a *comediator* term tuple

  $[\alpha_1, \alpha_2] : A_1 + A_2 \rightharpoonup B$

  (see Diagram 1) from the binary coproduct of the underlying diagram (pair of variable subsets) to the opvertex of the cocone. This is the unique term tuple, which commutes with opfirst and opsecond. We define this by using the mediator of the underlying instance cone and the comediator of the underlying type cocone. Existence and uniqueness represents the universality of the binary coproduct operator.

```
    (15) (SET.FTN$function comediator)
        (= (SET.FTN$source comediator) lang$language)
        (= (SET.FTN$target comediator) set.ftn$function)
        (= (SET.FTN$composition [comediator set.ftn$source]) cocone)
        (= (SET.FTN$composition [comediator set.ftn$target]) lang.tpl$tuple)
        (forall (?l (lang$language ?l))
```

```
(and (= (set.ftn$composition [(comediator ?l) (lang.tpl$index ?l)])
        (set.ftn$composition [(cocone-diagram ?l) (colimit ?l)]))
     (= (set.ftn$composition [(comediator ?l) (lang.tpl$arity ?l)])
        (opvertex ?l))))
     (forall (?s (cocone ?s))
         (and (= (lang.tpl$composition
                     [((injection1 ?l) (cocone-diagram ?s)) ((comediator ?l) ?s)])
                ((opfirst ?l) ?s))
              (= (lang.tpl$composition
                     [((injection2 ?l) (cocone-diagram ?s)) ((comediator ?l) ?s)])
                ((opsecond ?l) ?s))))))
```

Equational Presentations

`lang.eqn`

This section represents and axiomatizes universal algebraic equational presentations.

- For any type language ***L***, an ***L****-equation* is a doubleton $\{\alpha_1, \alpha_2\}$ consisting of ***L***-terms that share a common arity and return. Each equation has two components *terms*, and an *arity* and a return *type* that are shared with the component terms. IFF equations are regarded as a special case of IFF relations.

```
(1) (SET.FTN$function equation)
    (= (SET.FTN$source equation) language)
    (= (SET.FTN$target equation) set$set)

(2) (SET.FTN$function arity)
    (= (SET.FTN$source arity) language)
    (= (SET.FTN$target arity) set.ftn$function)
    (= (SET.FTN$composition [arity set.ftn$source]) equation)
    (= (SET.FTN$composition [arity set.ftn$target])
       (SET.FTN$composition [lang$variable set$power]))

(3) (SET.FTN$function type)
    (= (SET.FTN$source type) language)
    (= (SET.FTN$target type) set.ftn$function)
    (= (SET.FTN$composition [type set.ftn$source]) equation)
    (= (SET.FTN$composition [type set.ftn$target]) lang$entity)

(4) (SET.FTN$function term1)
    (= (SET.FTN$source term1) language)
    (= (SET.FTN$target term1) set.ftn$function)
    (= (SET.FTN$composition [term1 set.ftn$source]) equation)
    (= (SET.FTN$composition [term1 set.ftn$target]) lang.term$term)
    (forall (?l (lang$language ?l))
        (and (= (set.ftn$composition [(term1 ?l) (lang.term$arity ?l)]) (arity ?l))
             (= (set.ftn$composition [(term1 ?l) (lang.term$type ?l)]) (type ?l))

(5) (SET.FTN$function term2)
    (= (SET.FTN$source term2) language)
    (= (SET.FTN$target term2) set.ftn$function)
    (= (SET.FTN$composition [term2 set.ftn$source]) equation)
    (= (SET.FTN$composition [term2 set.ftn$target]) lang.term$term)
    (forall (?l (lang$language ?l))
        (and (= (set.ftn$composition [(term2 ?l) (lang.term$arity ?l)]) (arity ?l))
             (= (set.ftn$composition [(term2 ?l) (lang.term$type ?l)]) (type ?l))
```

- An (extended) type language – an *equational presentation* – is a pair $\langle \boldsymbol{L}, E \rangle$, where ***L*** is a type language and E is a set of ***L***-equations.

```
(6) (SET$class equational-presentation)

(7) (SET.FTN$function language)
    (= (SET.FTN$source language) equational-presentation)
    (= (SET.FTN$target language) lang$language)

(8) (SET.FTN$function equation-set)
    (= (SET.FTN$source equation-set) equational-presentation)
    (= (SET.FTN$target equation-set) set$set)

(9) (forall (?ep (equational-presentation ?ep))
        (set$subset (equation-set ?ep) (equation (language ?ep))))

language : equational-presentation → language
equation : language → set
power : set → set
```

The Namespace of Algebras

Table 1 lists the terminology for algebras and homomorphisms.

Table 1: Terminology for algebras and homomorphisms

	Function	Other
lang.alg	algebra domain operation	
	tuple-set domain-subset operation-term operation-cocone operation-tuple	
	binary-product-diagram binary-product-cone	
lang.alg.mor	homomorphism source target component	
	tuple-set-morphism component-subset	
	binary-product-diagram	
	composition identity	composition-opspan composable
lang.eqn.alg	satisfies algebra	

Algebras

lang.alg

The algebras for a function type language ***L*** in the IFF Algebraic Theory Ontology (IFF-AT) represent many-sorted universal algebra. Examples include monoids, groups, rings, modules, and automata.

- For any type language ***L***, an ***L****-algebra* consists of (Diagram 1)
  - a *domain* class function $dom(\boldsymbol{A}) : ent(\boldsymbol{L}) \to \mathsf{set}$, and
  - an *operation* class function $opr(\boldsymbol{A}) : ftn(\boldsymbol{L}) \to \mathsf{ftn}$.

  Note the asymmetry in the source collections of the domain versus the operation functions: the domain function is defined on entity types, whereas the operation function is defined on indicia, which are subsets of variable – names for entity types. Of course, indicia and signatures are equivalent.

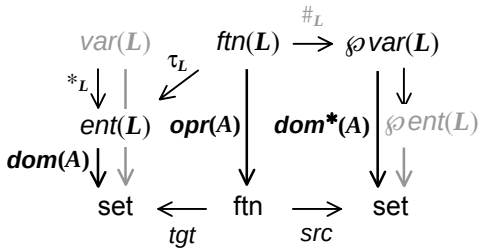

Diagram 1: Algebra – basic version

```
(1) (KIF$function algebra)
    (= (KIF$source algebra) lang$language)
    (= (KIF$target algebra) SET$class)

(2) (KIF$function domain)
    (= (KIF$source domain) lang$language)
    (= (KIF$source domain) KIF$function)
    (forall (?l (lang$language ?l))
        (and (= (KIF$source (domain ?l)) (algebra ?l))
             (= (KIF$target (domain ?l)) SET.FTN$function)
             (forall (?a ((algebra ?l) ?a))
                 (and (= (SET.FTN$source ((domain ?l) ?a)) (lang$entity ?l))
                      (= (SET.FTN$target ((domain ?l) ?a)) set$set)))))

(3) (KIF$function operation)
    (= (KIF$source operation) lang$language)
    (= (KIF$source operation) KIF$function)
    (forall (?l (lang$language ?l))
        (and (= (KIF$source (operation ?l)) (algebra ?l))
             (= (KIF$target (operation ?l)) SET.FTN$function)
             (forall (?a ((algebra ?l) ?a))
                 (and (= (SET.FTN$source ((operation ?l) ?a)) (lang$function ?l))
                      (= (SET.FTN$target ((operation ?l) ?a)) set.ftn$function)
                      (= (SET.FTN$composition [((operation ?l) ?a) set.ftn$source])
                         (SET.FTN$composition
                             [(lang$arity ?l)
                             (SET.FTN$composition
                                 [(set.ftn$power (lang$reference ?l))
                                  ((domain-subset ?l) ?a)])]))
                      (= (SET.FTN$composition [((operation ?l) ?a) set.ftn$target])
                         (SET.FTN$composition
                             [(lang$return ?l)
                              (SET.FTN$composition
                                  [(lang$reference ?l) ((domain ?l) ?a)])])))))))
```

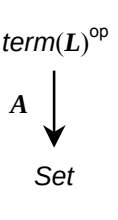

Figure 1a: Algebra – abstract

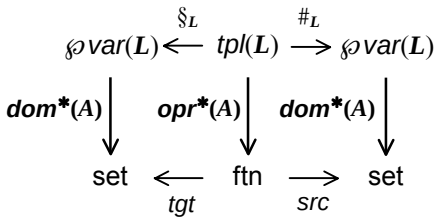

Diagram 2: Algebra – details

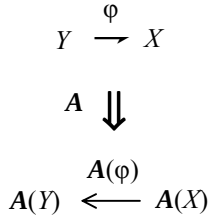

Figure 1b: Algebra – picture

- By extension, an ***L**-algebra* (Figure 1, Diagram 2) is a contravariant functor

  $\boldsymbol{A} : \mathit{term}(\boldsymbol{L})^{\mathrm{op}} \to \mathsf{Set}$

  that preserves products $\boldsymbol{A}(X_0+X_1) = \boldsymbol{A}(X_0) \times \boldsymbol{A}(X_1)$.

  This functor consists of

  – a *domain* class function on *indicia* (subsets of variables)

  $\mathit{dom}^*(\boldsymbol{A}) : \wp \mathit{var}(\boldsymbol{L}) \to \mathsf{set}$, and

  – an *operation* class function on term *tuples*

  $\mathit{opr}^*(\boldsymbol{A}) : \mathit{tpl}(\boldsymbol{L}) \to \mathsf{ftn}$.

  These class functions are compatible with respect to the source and target class functions for the category Set in the sense that Diagram 2 is commutative.

  In order to construct the functor two extensions are necessary – one for the object function of the functor ***dom***(**A**)* and one for the morphism function of the functor ***opr***(**A**)*. In addition, the morphism extension proceeds in two steps, first extend to terms and then extend to term tuples.

- The domain function $\mathit{dom}(\boldsymbol{A})$ is extended to $\mathit{dom}^*(\boldsymbol{A})$ using three properties:

  – The domain of the initial indicia $\varnothing$ is the *terminal* set

  $\mathit{dom}^*(\boldsymbol{A})(\varnothing) = 1.$

  – The domain of the binary coproduct of indicia is the binary Cartesian *product*

  $\mathit{dom}^*(\boldsymbol{A})(X_0+X_1) = \mathit{dom}^*(\boldsymbol{A})(X_0) \times \mathit{dom}^*(\boldsymbol{A})(X_1).$

  – The domain of a singleton indicia is a *basic* domain

  $\mathit{dom}^*(\boldsymbol{A})(\{x\}) = \mathit{dom}(\boldsymbol{A})(*_{\boldsymbol{L}}(x)).$

  – In general, the domain of any indicia $X \subseteq \mathit{var}(\boldsymbol{L})$ is the Cartesian product of the basic domains of its elements

  $\mathit{dom}^*(\boldsymbol{A})(X) = \prod_{x \in X} \mathit{dom}(\boldsymbol{A})(*_{\boldsymbol{L}}(x)).$

```
(4) (KIF$function tuple-set)
    (= (KIF$source tuple-set) lang$language)
    (= (KIF$source tuple-set) KIF$function)
    (forall (?l (lang$language ?l))
        (and (= (KIF$source (tuple-set ?l)) (algebra ?l))
             (= (KIF$target (tuple-set ?l)) KIF$function)
             (forall (?a ((algebra ?l) ?a))
                 (and (= (KIF$source ((tuple-set ?l) ?a)) (set$power (lang$entity ?l)))
                      (= (KIF$target ((tuple-set ?l) ?a)) set.lim.prd$tuple-set)
                      (forall (?x (set$subset ?x (lang$variable ?l)))
                          (and (= (set.lim.prd$index (((tuple-set ?l) ?a) ?x)) ?x)
                               (SET.FTN$restriction
                                   (((tuple-set ?l) ?a) ?x)
                                   (SET.FTN$composition [(reference ?l) ((domain ?l) ?a)])))))))))

(5) (KIF$function domain-subset)
    (= (KIF$source domain-subset) lang$language)
    (= (KIF$source domain-subset) KIF$function)
    (forall (?l (lang$language ?l))
```

```
        (and (= (KIF$source (domain-subset ?l)) (algebra ?l))
             (= (KIF$target (domain-subset ?l)) SET.FTN$function)
             (forall (?a ((algebra ?l) ?a))
                 (and (= (SET.FTN$source ((domain-subset ?l) ?a)) (set$power (lang$variable ?l)))
                      (= (SET.FTN$target ((domain-subset ?l) ?a)) set$set)
                      (= ((domain-subset ?l) ?a)
                         (set.ftn$composition [((tuple-set ?l) ?a) set.lim.prd$product])))))))
```

- The basic operation function $opr(\boldsymbol{A})$ is first extended to an operation function on terms $opr^{+}(\boldsymbol{A})$:
  - Consider an elementary term $\in_{\tau, x}$ built from a case (τ, x) consisting of an signature $\tau \in sign(*_{\boldsymbol{L}})$ and a variable $x \in arity(\boldsymbol{L})(\tau) = X$ that references an entity type $refer(\boldsymbol{L})(x) = \alpha$. The operation for the elementary term $\in_{\tau, x}$ is the *projection* function
    $$opr^{+}(\boldsymbol{A})(\in_{\tau, x}) = \pi(\boldsymbol{A})(X) : dom^{*}(\boldsymbol{A})(X) \to dom(\boldsymbol{A})(*_{\boldsymbol{L}}(x)) = dom(\boldsymbol{A})(\alpha).$$
  - Consider any substitutable pair (f, φ) consisting of a function type $f \in ftn(\boldsymbol{L})$ and a term tuple $\varphi \in term^{*}(\boldsymbol{L})$, where the function arity of f is the same as the index of φ. The operation for the substitution term $f[\varphi] : refer\text{-}sign(\boldsymbol{L})(A) \rhd \alpha$ is the function *composition* of the operation for the term tuple φ followed by the operation for the function type f
    $$opr^{+}(\boldsymbol{A})(f[\varphi]) = opr^{*}(\boldsymbol{A})(\varphi) \cdot opr(\boldsymbol{A})(f).$$

```
(6) (KIF$function operation-term)
    (= (KIF$source operation-term) lang$language)
    (= (KIF$source operation-term) KIF$function)
    (forall (?l (lang$language ?l))
        (and (= (KIF$source (operation-term ?l)) (algebra ?l))
             (= (KIF$target (operation-term ?l)) SET.FTN$function)
             (forall (?a ((algebra ?l) ?a))
                 (and (= (SET.FTN$source ((operation-term ?l) ?a)) (lang.term$term ?l))
                      (= (SET.FTN$target ((operation-term ?l) ?a)) set.ftn$function)
                      (= (SET.FTN$composition [((operation-term ?l) ?a) set.ftn$source])
                         (SET.FTN$composition [(lang.term$arity ?l) ((domain-subset ?l) ?a)]))
                      (= (SET.FTN$composition [((operation-term ?l) ?a) set.ftn$target])
                         (SET.FTN$composition [(lang.term$type ?l) ((domain ?l) ?a)]))
                      (=> ((lang.term$elementary ?l) ?t)
                          (= (((operation-term ?l) ?a) ?t)
                             ((set.lim.prd$projection
                                  (((tuple-set ?l) ?a) ((lang.term$indicia ?l) ?t)))
                              ((lang.term$variable ?l) ?t)))
                      (=> ((lang.term$composite ?l) ?t)
                          (= (((operation-term ?l) ?a) ?t)
                             (set.ftn$composition
                                 [(((operation-tuple ?l) ?a) ((lang.term$tuple ?l) ?t))
                                  (((operation ?l) ?a) ((lang.term$function ?l) ?t))])))))))))
```

- The operation function $opr(\boldsymbol{A})$ is next extended to the operation function on term tuples $opr^{*}(\boldsymbol{A})$. The following operations can be conceived for the three term tuple constructors.
  - Consider an indicia X. The operation for the empty term tuple
    $$0_X : \varnothing \multimap X$$
    is the *unique* function from the domain at indicia X to the terminal set.
    $$opr^{*}(\boldsymbol{A})(0_X) = !_{dom_{*}(\boldsymbol{A})(X)} : dom^{*}(\boldsymbol{A})(X) \to 1.$$
  - Consider tuplable term tuples $\varphi_0 : X_0 \multimap X$ and $\varphi_1 : X_1 \multimap X$. The operation for the binary term tupling
    $$[\varphi_0, \varphi_1] : A_0 + A_1 \multimap A$$
    is the Cartesian product pairing of the component operations
    $$opr^{*}(\boldsymbol{A})([\varphi_0, \varphi_1]) = [opr^{*}(\boldsymbol{A})(\varphi_0), opr^{*}(\boldsymbol{A})(\varphi_1)] : dom^{*}(\boldsymbol{A})(X) \to dom^{*}(\boldsymbol{A})(X_0) \times dom^{*}(\boldsymbol{A})(X_1).$$
  - Consider an indexed term (x, t) consisting of a variable $x \in var(\boldsymbol{L})$ and a term $t : A \rhd \alpha$ with $refer(\boldsymbol{L})(x) = \alpha = typ(\boldsymbol{L})(t)$. The operation for the singleton term tuple
    $$\{(x, t)\} : \{x\} \multimap X$$
    is the *basic* operation on the term
    $$opr^{*}(\boldsymbol{A})(\{(x, t)\}) = opr^{+}(\boldsymbol{A})(t) : dom^{*}(\boldsymbol{A})(X) \to dom(\boldsymbol{A})(\alpha).$$

One problem with giving a recursive definition following the above three constructors is that the second case needs a selector that uses a partition of the index of a term tuple: "If t is a term tuple and p is a partition on its index, … ." So instead we define the tuple operation function in one full swoop.

– Consider any term tuple $\varphi \in \mathit{tpl}(\boldsymbol{L})$. Define a Cartesian product cone, whose tuple-set is the domain tuple set (see above) for indicia $\mathit{index}(\boldsymbol{L})(\varphi)$, whose vertex is the subset domain $\mathit{dom}^*(\boldsymbol{A})(\mathit{arity}(\boldsymbol{L})(\varphi))$, and whose y^{th} component function index by the variable $y \in \mathit{index}(\boldsymbol{L})(\varphi)$, where $\mathit{refer}(\boldsymbol{L})(x) = \alpha$, is the operation on the selection term

$$\mathit{opr}^+(\boldsymbol{A})(\sigma_x(\varphi)) : \mathit{dom}^*(\boldsymbol{A})(\mathit{arity}(\boldsymbol{L})(\varphi)) \to \mathit{dom}(\boldsymbol{A})(\alpha).$$

The operation for the term tuple

$$\varphi : Y \rightharpoonup X$$

is the mediator of the Cartesian product cone of the selection operations indexed by variables in the index of a term tuple

$$\mathit{opr}^*(\boldsymbol{A})(\varphi) = [\mathit{opr}^+(\boldsymbol{A})(\sigma_x(\varphi))]_{y \in Y} : \mathit{dom}^*(\boldsymbol{A})(X) \to \mathit{dom}^*(\boldsymbol{A})(Y) = \textstyle\prod_{y \in Y} \mathit{dom}(\boldsymbol{A})(*_{\boldsymbol{L}}(y)).$$

```
(7) (KIF$function operation-cocone)
    (= (KIF$source operation-cocone) lang$language)
    (= (KIF$source operation-cocone) KIF$function)
    (forall (?l (lang$language ?l))
        (and (= (KIF$source (operation-cocone ?l)) (algebra ?l))
             (= (KIF$target (operation-cocone ?l)) KIF$function)
             (forall (?a ((algebra ?l) ?a))
                 (and (= (KIF$source ((operation-cocone ?l) ?a)) lang.tpl$tuple)
                      (= (KIF$target ((operation-cocone ?l) ?a)) set.lim.prd$cocone)
                      (forall (?t ((tuple ?l) ?t))
                          (and (= (set.lim.prd$cone-tuple-set (((operation-cocone ?l) ?a) ?t))
                                  (((tuple-set ?l) ?a) ((lang.tpl$index ?l) ?t)))
                               (= (set.lim.prd$vertex (((operation-cocone ?l) ?a) ?t))
                                  (((domain-subset ?l) ?a) ((lang.term$arity ?l) ?t)))
                               (forall (?x (((lang.tpl$index ?l) ?t) ?x))
                                   (= ((set.lim.prd$component (((operation-cocone ?l) ?a) ?t)) ?x)
                                      (((operation-tuple ?l) ?a) ((selection ?l) [?t ?x]))))))))))))

(8) (KIF$function operation-tuple)
    (= (KIF$source operation-tuple) lang$language)
    (= (KIF$source operation-tuple) KIF$function)
    (forall (?l (lang$language ?l))
        (and (= (KIF$source (operation-tuple ?l)) (algebra ?l))
             (= (KIF$target (operation-tuple ?l)) SET.FTN$function)
             (forall (?a ((algebra ?l) ?a))
                 (and (= (SET.FTN$source ((operation-tuple ?l) ?a)) (lang.tpl$tuple ?l))
                      (= (SET.FTN$target ((operation-tuple ?l) ?a)) set.ftn$function)
                      (= (SET.FTN$composition [((operation-tuple ?l) ?a) set.ftn$source])
                         (SET.FTN$composition [(lang.tpl$arity ?l) ((domain-subset ?l) ?a)]))
                      (= (SET.FTN$composition [((operation-tuple ?l) ?a) set.ftn$target])
                         (SET.FTN$composition [(lang.tpl$index ?l) ((domain-subset ?l) ?a)]))
                      (= ((operation-tuple ?l) ?a)
                         (set.ftn$composition [((operation-cocone ?l) ?a) set.lim.prd$mediator])))))))
```

○ To be functorial an algebra must preserve term tuple composition and term tuple identities. If $\psi, \varphi \in \mathit{term}^*(\boldsymbol{L})$ are two composable term tuples, then $\boldsymbol{A}(\psi \circ \varphi) = \boldsymbol{A}(\varphi) \cdot \boldsymbol{A}(\psi)$. If $X \subseteq \mathit{var}(\boldsymbol{L})$ is an indicia, then $\boldsymbol{A}(\mathit{id}_X) = \mathit{id}_{\boldsymbol{A}(X)}$. These properties are derivable from the operation definitions above.

```
(4) (forall (?l (lang$language ?l) ?a ((algebra ?l) ?a)
             ?t2 ((lang.tpl$tuple ?l) ?t2)?t1 ((lang.tpl$tuple ?l) ?t1)
             (lang.tpl$composable ?t2 ?t1))
        (= (((operation-tuple ?l) ?a) ((lang.tpl$composition ?l) [?t2 ?t1]))
           (set.ftn$composition
               [(((operation-tuple ?l) ?a) ?t1) (((operation-tuple ?l) ?a) ?t2)])))

(5) (forall (?l (lang$language ?l) ?a ((algebra ?l) ?a)
             ?x (set$subset ?x (lang$variable ?l)))
        (= (((operation-tuple ?l) ?a) ((lang.tpl$identity ?l) ?x))
           (set.ftn$identity (((domain-subset ?l) ?a) ?x))))
```

- An algebra maps the initial subset of variables to the terminal set, and it maps binary coproducts to binary products. These properties are derivable from the operation definitions above. To express the latter property, we need to define two auxiliary functions: a function that maps binary coproduct diagrams of term tuples to binary product diagrams of sets and functions; and a function that maps binary coproduct cocones of term tuples to binary product cones of sets and functions. Hence, an algebra (being contravariant) preserves finite products.

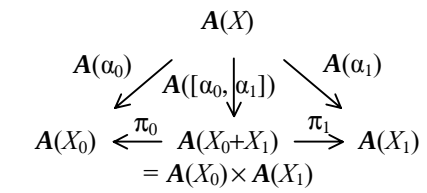

Diagram 1: Algebras preserving binary products

```
(6) (forall (?l (lang$language ?l) ?a ((algebra ?l) ?a))
        (and (= (((domain-subset ?l) ?a) (lang.tpl.col$initial ?l)) set.lim$terminal)
             (= (SET.FTN$composition [(lang.tpl.colim$counique ?l) ((operation-tuple ?l) ?a)])
                (SET.FTN$composition [((domain-subset ?l) ?a) set.lim$unique])))))

(7) (KIF$function binary-product-diagram)
    (= (KIF$source binary-product-diagram) lang$language)
    (= (KIF$source binary-product-diagram) KIF$function)
    (forall (?l (lang$language ?l))
        (and (= (KIF$source (binary-product-diagram ?l)) (algebra ?l))
             (= (KIF$target (binary-product-diagram ?l)) SET.FTN$function)
             (forall (?a ((algebra ?l) ?a))
                 (and (= (SET.FTN$source ((binary-product-diagram ?l) ?a))
                         (lang.tpl.colim.coprd2$diagram ?l))
                      (= (SET.FTN$target ((binary-product-diagram ?l) ?a)) set.lim.prd2$diagram)
                      (= (SET.FTN$composition [((binary-product-diagram ?l) ?a) set.lim.prd2$set1])
                         (SET.FTN$composition
                             [(lang.tpl.colim.coprd2$subset1 ?l) ((domain-subset ?l) ?a)]))
                      (= (SET.FTN$composition [((binary-product-diagram ?l) ?a) set.lim.prd2$set2])
                         (SET.FTN$composition
                             [(lang.tpl.colim.coprd2$subset2 ?l) ((domain-subset ?l) ?a)])))))))

(8) (KIF$function binary-product-cone)
    (= (KIF$source binary-product-cone) lang$language)
    (= (KIF$source binary-product-cone) KIF$function)
    (forall (?l (lang$language ?l))
        (and (= (KIF$source (binary-product-cone ?l)) (algebra ?l))
             (= (KIF$target (binary-product-cone ?l)) SET.FTN$function)
             (forall (?a ((algebra ?l) ?a))
                 (and (= (SET.FTN$source ((binary-product-cone ?l) ?a))
                         (lang.tpl.colim.coprd2$cocone ?l))
                      (= (SET.FTN$target ((binary-product-cone ?l) ?a)) set.lim.prd2$cone)

                      (= (SET.FTN$composition
                             [((binary-product-cone ?l) ?a) set.lim.prd2$cone-diagram])
                         (SET.FTN$composition
                             [(lang.tpl.colim.coprd2$cocone-diagram ?l)
                              ((binary-product-diagram ?l) ?a)]))
                      (= (SET.FTN$composition [((binary-product-cone ?l) ?a) set.lim.prd2$vertex])
                         (SET.FTN$composition
                             [(lang.tpl.colim.coprd2$opvertex ?l) ((domain-subset ?l) ?a)]))
                      (= (SET.FTN$composition [((binary-product-cone ?l) ?a) set.lim.prd2$first])
                         (SET.FTN$composition
                             [(lang.tpl.colim.coprd2$opfirst ?l) ((operation-tuple ?l) ?a)]))
                      (= (SET.FTN$composition [((binary-product-cone ?l) ?a) set.lim.prd2$second])
                         (SET.FTN$composition
                             [(lang.tpl.colim.coprd2$opsecond ?l) ((operation-tuple ?l) ?a)])))))))

(9) (forall (?l (lang$language ?l) ?a ((algebra ?l) ?a))
      (and (= (SET.FTN$composition
                  [(lang.tpl.colim.coprd2$binary-coproduct ?l) ((domain-subset ?l) ?a)])
              (SET.FTN$composition [((binary-product-diagram ?l) ?a) set.lim.prd2$binary-product]))
           (= (SET.FTN$composition [(lang.tpl.colim.coprd2$injection1 ?l) ((operation-tuple ?l) ?a)])
              (SET.FTN$composition [((binary-product-diagram ?l) ?a) set.lim.prd2$projection1]))
           (= (SET.FTN$composition [(lang.tpl.colim.coprd2$injection2 ?l) ((operation-tuple ?l) ?a)])
              (SET.FTN$composition [((binary-product-diagram ?l) ?a) set.lim.prd2$projection2]))
           (= (SET.FTN$composition [(lang.tpl.colim.coprd2$comediator ?l) ((operation-tuple ?l) ?a)])
              (SET.FTN$composition [((binary-product-cone ?l) ?a) set.lim.prd2$mediator])))))
```

Homomorphisms

`lang.alg.mor`

The homomorphisms for a function type language ***L*** in the IFF Algebraic Theory Ontology (IFF-AT) represent many-sorted universal algebraic homomorphisms. Examples include monoid morphisms, group, ring and module homomorphisms, and morphisms of automata.

- For any type language ***L***, an ***L****-homomorphism* $\boldsymbol{h} : \boldsymbol{A}_1 \to \boldsymbol{A}_2$ from source ***L***-algebra $\boldsymbol{A}_1$ to target ***L***-algebra $\boldsymbol{A}_2$ consists of
  - a *component* function $cmpt(\boldsymbol{h}) : ent(\boldsymbol{L}) \to \mathsf{ftn}$.

  This function commutes with the operations in the sense of Diagram 1. In order to express this the component function needs to be extended from entity types to indicia.

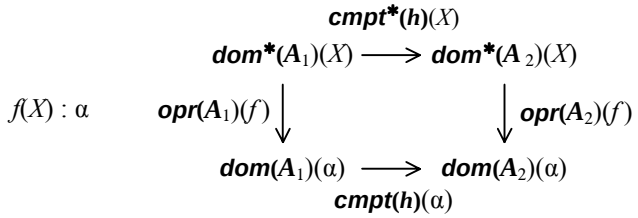

Diagram 1: Homomorphism – basic version

```
(1) (KIF$function homomorphism)
    (= (KIF$source homomorphism) lang$language)
    (= (KIF$target homomorphism) SET$class)

(2) (KIF$function source)
    (= (KIF$source source) lang$language)
    (= (KIF$source source) SET.FTN$function)
    (forall (?l (lang$language ?l))
        (and (= (SET.FTN$source (source ?l)) (homomorphism ?l))
             (= (SET.FTN$target (source ?l)) (algebra ?l))))

(3) (KIF$function target)
    (= (KIF$source target) lang$language)
    (= (KIF$source target) SET.FTN$function)
    (forall (?l (lang$language ?l))
        (and (= (SET.FTN$source (target ?l)) (homomorphism ?l))
             (= (SET.FTN$target (target ?l)) (algebra ?l))))

(4) (KIF$function component)
    (= (KIF$source component) lang$language)
    (= (KIF$source component) KIF$function)
    (forall (?l (lang$language ?l))
        (and (= (KIF$source (component ?l)) (homomorphism ?l))
             (= (KIF$target (component ?l)) SET.FTN$function)
             (forall (?h ((homomorphism ?l) ?h))
                 (and (= (SET.FTN$source ((component ?l) ?h)) (lang$entity ?l))
                      (= (SET.FTN$target ((component ?l) ?h)) set.ftn$function)
                      (= (SET.FTN$composition [((component ?l) ?h) set.ftn$source])
                         ((domain ?l) (source ?h)))
                      (= (SET.FTN$composition [((component ?l) ?a) set.ftn$target])
                         ((domain ?l) (target ?h)))
                      (forall (?f ((lang$function ?l) ?f))
                          (= (set.ftn$composition
                                 [(((component-subset ?l) ?a) ((lang$function-arity ?l) ?f))
                                  (((operation ?l) (target ?h)) ?f)])
                             (set.ftn$composition
                                 [(((operation ?l) (source ?h)) ?f)
                                  (((component ?l) ?a) ((lang$type ?l) ?f))]))))))))
```

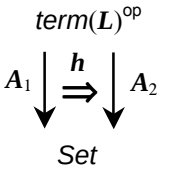

Figure 1: Homomorphism – abstract

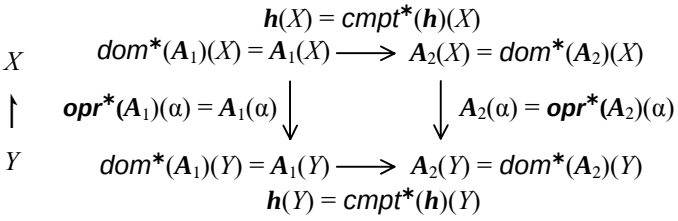

Diagram 2: Homomorphism – details

- By extension, an ***L****-homomorphism* $\boldsymbol{h} : \boldsymbol{A}_1 \to \boldsymbol{A}_2$ (Figure 1, Diagram 2) from ***L***-algebra $\boldsymbol{A}_1$ to ***L***-algebra $\boldsymbol{A}_2$ is a natural transformation

  $\boldsymbol{h} : \boldsymbol{A} \Rightarrow \boldsymbol{A} : \mathit{term}(\boldsymbol{L})^{\mathrm{op}} \to \mathsf{Set}$

  from contravariant functor $\boldsymbol{A}_1$ to contravariant functor $\boldsymbol{A}_2$. This natural transformation consists of

  – a *component* function $\mathit{cmpt}(\boldsymbol{h}) : \wp \mathit{var}(\boldsymbol{L}) \to \mathsf{ftn}$

  that maps every indicia variable $X \subseteq \mathit{var}(\boldsymbol{L})$ to an extended component function

  $\mathit{cmpt}^*(\boldsymbol{h})(X) = \prod_{x \in X} \mathit{cmpt}(\boldsymbol{h})(*_{\boldsymbol{L}}(x)) : \mathit{dom}^*(\boldsymbol{A}_1)(X) \to \mathit{dom}^*(\boldsymbol{A}_2)(X) = \prod_{x \in X} \mathit{dom}(\boldsymbol{A}_2)(*_{\boldsymbol{L}}(x))$.

  This component class function is compatible with the extended domain functions of the source and target algebras. It commutes with the extended operation functions of the source and target algebras, satisfying the naturality Diagram 2. To axiomatize this we use the notion of a morphism of tuple-sets, specifically a morphisms between the tuple-sets of the source and target algebras.

```
(5) (KIF$function tuple-set-morphism)
    (= (KIF$source tuple-set-morphism) lang$language)
    (= (KIF$source tuple-set-morphism) KIF$function)
    (forall (?l (lang$language ?l))
        (and (= (KIF$source (tuple-set-morphism ?l)) (homomorphism ?l))
             (= (KIF$target (tuple-set-morphism ?l)) KIF$function)
             (forall (?h ((homomorphism ?l) ?h))
                 (and (= (KIF$source ((tuple-set-morphism ?l) ?h)) (set$power (lang$entity ?l)))
                      (= (KIF$target ((tuple-set-morphism ?l) ?h)) set.lim.prd$tuple-set-morphism)
                      (forall (?x (set$subset ?x (lang$variable ?l)))
                          (and (= (set.lim.prd$source (((tuple-set-morphism ?l) ?h) ?x))
                                  (((tuple-set-morphism ?l) (source ?h)) ?x))
                               (= (set.lim.prd$target (((tuple-set-morphism ?l) ?h) ?x))
                                  (((tuple-set-morphism ?l) (target ?h)) ?x))
                               (= (set.lim.prd$index (((tuple-set-morphism ?l) ?h) ?x))
                                  (set.ftn$identity
                                      (set.lim.prd$index (((tuple-set ?l) (source ?h)) ?x))))
                               (SET.FTN$restriction
                                   (((tuple-set-morphism ?l) ?h) ?x)
                                   (SET.FTN$composition [(reference ?l) ((component ?l) ?h)])))))))))

(6) (KIF$function component-subset)
    (= (KIF$source component-subset) lang$language)
    (= (KIF$source component-subset) KIF$function)
    (forall (?l (lang$language ?l))
        (and (= (KIF$source (component-subset ?l)) (homomorphism ?l))
             (= (KIF$target (component-subset ?l)) SET.FTN$function)
             (forall (?h ((homomorphism ?l) ?h))
                 (and (= (SET.FTN$source ((component-subset ?l) ?h)) (set$power (lang$variable ?l)))
                      (= (SET.FTN$target ((component-subset ?l) ?h)) set.ftn$function)
                      (forall (?x (set$subset ?x (lang$variable ?l)))
                          (= (((component-subset ?l) ?h) ?x)
                             (set.lim.prd.mor$product (((tuple-set-morphism ?l) ?h) ?x)))))))))
```

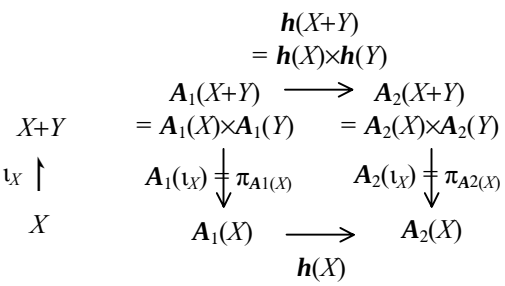

Diagram 1: Homomorphisms preserving binary products

- A homomorphism maps the initial indicia to the identity function on the terminal set, and it maps binary coproducts of a pair of indicia to the binary product of the component functions. To axiomatize the latter property, we need to define an auxiliary function: a function that maps binary coproduct diagrams of term tuples to binary product diagrams of functions.

```
(7) (forall (?l (lang$language ?l) ?h ((homomorphism ?l) ?h))
        (= (((component ?l) ?h) (lang.tpl.colim$initial ?l))
           (set.ftn$identity set.lim$terminal)))

(8) (KIF$function binary-product-diagram)
    (= (KIF$source binary-product-diagram) lang$language)
    (= (KIF$source binary-product-diagram) KIF$function)
    (forall (?l (lang$language ?l))
        (and (= (KIF$source (binary-product-diagram ?l)) (homomorphism ?l))
             (= (KIF$target (binary-product-diagram ?l)) SET.FTN$function)
             (forall (?h ((homomorphism ?l) ?h))
                 (and (= (SET.FTN$source ((binary-product-diagram ?l) ?h))
                         (lang.tpl.colim.coprd2$diagram ?l))
                      (= (SET.FTN$target ((binary-product-diagram ?l) ?h))
                         set.lim.prd2.ftn$diagram)
                      (= (SET.FTN$composition
                             [((binary-product-diagram ?l) ?h) set.lim.prd2.ftn$function1])
                         (SET.FTN$composition
                             [(lang.tpl.colim.coprd2$subset1 ?l)
                              ((component-subset ?l) ?h)]))
                      (= (SET.FTN$composition
                             [((binary-product-diagram ?l) ?h) set.lim.prd2.ftn$function2])
                         (SET.FTN$composition
                             [(lang.tpl.colim.coprd2$subset2 ?l)
                              ((component-subset ?l) ?h)])))))))

(9) (forall (?l (lang$language ?l) ?h ((homomorphism ?l) ?h))
        (= (SET.FTN$composition
               [(lang.tpl.colim.coprd2$binary-product ?l) ((component ?l) ?h)])
           (SET.FTN$composition
               [((binary-product-diagram ?l) ?h) set.lim.prd2.ftn$binary-product])))
```

- Two homomorphisms are *composable* when the target of the first is equal to the source of the second. The *composition* of two composable homomorphisms $\boldsymbol{h}_1 : \boldsymbol{A} \to \boldsymbol{A}'$ and $\boldsymbol{h}_2 : \boldsymbol{A}' \to \boldsymbol{A}''$ is defined in terms of the composition of their component functions.

```
(10) (SET.LIM.PBK$opspan composable-opspan)
     (= (SET.LIM.PBK$class1 composable-opspan) homomorphism)
     (= (SET.LIM.PBK$class2 composable-opspan) homomorphism)
     (= (SET.LIM.PBK$opvertex composable-opspan) lang.alg$algebra)
     (= (SET.LIM.PBK$first composable-opspan) target)
     (= (SET.LIM.PBK$second composable-opspan) source)

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

(12) (SET.FTN$function composition)
```

```
(= (SET.FTN$source composition) (SET.LIM.PBK$pullback composable-opspan))
(= (SET.FTN$target composition) homomorphism)
(forall (?h1 (homomorphism ?h1) ?h2 (homomorphism ?h2) (composable ?h1 ?h2))
    (and (= (source (composition [?h1 ?h2])) (source ?h1))
         (= (target (composition [?h1 ?h2])) (target ?h2))
         (forall (?x (set$subset ?x (lang$variable ?l)))
             (= (((component-subset ?l) (composition [?h1 ?h2])) ?x)
                (set.ftn$composition
                    [(((component-subset ?l) ?h1) ?x)
                     (((component-subset ?l) ?h2) ?x)])))))
```

- Composition satisfies the usual *associative law*.

```
(forall (?h1 (homomorphism ?h1)
         ?h2 (homomorphism ?h2)
         ?h3 (homomorphism ?h3)
         (composable ?h1 ?h2) (composable ?h2 ?h3))
    (= (composition [?h1 (composition [?h2 ?h3])])
       (composition [(composition [?h1 ?h2]) ?h3])))
```

- For any hypergraph ***G***, there is an *identity* hypergraph morphism.

```
(13) (SET.FTN$function identity)
     (= (SET.FTN$source identity) lang.alg$algebra)
     (= (SET.FTN$target identity) homomorphism)
     (forall (?a (lang.alg$algebra ?a))
         (and (= (source (identity ?a)) ?a)
              (= (target (identity ?a)) ?a)
              (= ((component-subset ?l) (identity ?a))
                 (SET.FTN$composition [((domain-subset ?l) ?a) set.ftn$identity])))))
```

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

```
(forall (?h (homomorphism ?h))
    (and (= (composition [(identity (source ?h)) ?h]) ?h)
         (= (composition [?h (identity (target ?h))]) ?h)))
```

Varieties of Algebras

`lang.eqn.alg`

For any type language ***L***, an *equationally defined class* or *variety* of algebras is the subclass of ***L***-algebras that satisfies some equational presentation ⟨***L***, *E*⟩.

- For any type language ***L***, an ***L***-algebra ***A*** *satisfies* an ***L***-equation $\varepsilon = \{\alpha_1, \alpha_2\} \in eqn(\boldsymbol{L})$,

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

  when ε is true when interpreted in the context ***A***; that is, when the term operation function of ***A*** applied to the component terms of ε results in equal operations (set functions).

```
(1) (KIF$function satisfies)
    (= (KIF$source satisfies) lang$language)
    (= (KIF$source satisfies) SET.FTN$function)
    (forall (?l (lang$language ?l))
      (and (= (SET.FTN$source (satisfies ?l)) (algebra ?l))
           (= (SET.FTN$target (satisfies ?l)) set$set)
           (forall (?a ((algebra ?l) ?a))
             (and (set$subset ((satisfies ?l) ?a) (lang.eqn$equation ?l))
                  (forall (?e ((lang.eqn$equation ?l) ?e))
                    (<=> (((satisfies ?l) ?a) ?e)
                         (= (((lang.alg$operation-term ?l) ?a) ((lang.eqn$term1 ?l) ?e))
                            (((lang.alg$operation-term ?l) ?a) ((lang.eqn$term2 ?l) ?e)))))))))
```

- For any equational presentation pair ⟨***L***, *E*⟩, an ⟨***L***, *E*⟩-algebra ***A*** is an ***L***-algebra that *satisfies* every equation in *E*.

```
(2) (KIF$function algebra)
    (= (KIF$source algebra) lang.term.eqn$equational-presentation)
    (= (KIF$target algebra) SET$class)

(3) (forall (?ep (lang.term.eqn$equational-presentation ?ep))
        (and (CLS$subclass (algebra ?ep) (lang.alg$algebra (lang.term.eqn$language ?ep)))
             (forall ((lang.alg$algebra (language ?ep)) ?a))
                 (<=> (algebra ?ep)
                      (set$subset (lang.term.eqn$equation-set ?ep) ((satisfies ?l) ?a))))))
```

Algebraic Structures

`alg`

○ An *algebraic structure* $\langle \boldsymbol{L}, \boldsymbol{A} \rangle$ consists of a function type language $\boldsymbol{L}$ and an $\boldsymbol{L}$-algebra.

Combining Models and Algebraic Structures

○ In order to merge the notion of an IFF algebra with the notion of an IFF model we do the following.

– We identify $\mathsf{A}(A)$ with tuples of exact arity A : $\mathsf{A}(A) = \mathit{arity\text{-}fiber}(\boldsymbol{A})(A)$.

– We assume there is a monoid $\langle \mathit{tuple}(\boldsymbol{A}), \otimes_{\boldsymbol{A}}, 1_{\boldsymbol{A}} \rangle$ on tuples consisting of (1) a *concatenation* operation $\otimes_{\boldsymbol{A}} = \mathit{concat}(\boldsymbol{A}) : \mathit{tuple}(\boldsymbol{A}) \times \mathit{tuple}(\boldsymbol{A}) \rightarrow \mathit{tuple}(\boldsymbol{A})$ which maps any two tuples of arities A_0 and A_1 to a tuple of arity $A_0 + A_1$, and (2) an *empty* tuple $1_{\boldsymbol{A}} \in \mathit{tuple}(\boldsymbol{A})$ whose arity must be empty.

– We will also assume that concatenation on fibers restricts to a bijection

$$\mathsf{A}(A_0) \times \mathsf{A}(A_1) \cong \mathsf{A}(A_0 + A_1)$$

where the product is the Cartesian product.

$$\begin{array}{ccc} & & \mathit{arity}(\boldsymbol{A}) \times \mathit{arity}(\boldsymbol{A}) \\ \mathit{tuple}(\boldsymbol{A}) \times \mathit{tuple}(\boldsymbol{A}) & \longrightarrow & \wp \mathit{var}(\boldsymbol{A}) \times \wp \mathit{var}(\boldsymbol{A}) \\ \otimes_{\boldsymbol{A}} \downarrow & & \downarrow + \\ \mathit{tuple}(\boldsymbol{A}) & \xrightarrow[\mathit{arity}(\boldsymbol{A})]{} & \wp \mathit{var}(\boldsymbol{A}) \end{array}$$