The Portal of Conceptual Graphs

An IFF portal is point of entrance; it is a namespace that serves as an interface and is used for communication between an external representation and the IFF Model Theory Ontology.

The IFF Model Theory Ontology has a close and intimate relationship with conceptual graphs (without contexts). Here we give a preliminary discussion of that relationship. This section describes and axiomatizes the portal of conceptual graphs. The conceptual graphs portal is specifically oriented to the particular representation in the Conceptual Graph Standard, and the following axiomatization closely follows the discussion in that standard. The declarations in the portal of conceptual graphs use the lower metalevel terminology from the IFF Model Theory Ontology.

Table 1 lists the terminology for the IFF representation of conceptual graphs. Actual terms from the Conceptual Graphs Standard are in boldface. The other terms are needed in the IFF representation of conceptual graphs. The namespace in the magenta-colored background use only a limited notion of variable and define functions for primitive relation labels only – the basic CG namespace uses only natural numbers as variable, and the spangraph namespace uses cases (primitive relation label – natural number pairs) as variables. The namespace in the green-colored background have the full extension to variables and to all relation labels, primitive and defined.

Table 1: The terminology of the conceptual graphs portal

	Set	Function	Other
cg	**type-label** **= concept-label** **individual-marker** **primitive** **defined** **relation-label**	**valence** arity **signature**	**type-hierarchy** **= concept-hierarchy** **= subtype** **relation-hierarchy** **entity absurdity** pair hypergraph
cg .sgph	case	injection indication projection comediator	spangraph
		type arity signature	language
cg .lang	variable = coreference-label	primitive-arity **primitive-signature** **lambda** defined-arity defined-signature type arity signature	language
cg .expr	expression expression-pair expression-variable-pair expression-substitution-pair	arity signature	expression-language expression-classification
		atom negation *conjunction disjunction* *implication equivalence* *existential-quantification* *universal-quantification* *substitution*	
cg .interp		lambda-star	interpretation
cg .mod	tuple		entity-classification relation-classification **knowledge-base**

Hypergraphs

cg

Correspondences between the CG standard and IFF are listed in Table 2.

Table 2: Correspondences

CG	IFF
conceptual graph	classification incidence between a tuple and an expression
concept	classification incidence between a universe element (entity instance) and an entity type
type label	entity type
relation label	relation type
conceptual relation	tuple
coreference set	variable
referent	entity instance
relation arc	case (entity instance or variable)
entity arc	comediator

In this section we discuss how conceptual graphs define hypergraphs.

○ The entity *type hierarchy* is a partially ordered set whose elements are called *type labels*. Type labels are also called *concept labels*. We ignore defined type labels. We assume that all type labels are primitive.

```
(1) (set$set type-label)
    (set$set concept-label)
    (= concept-label type-label)
```

```
(2) (ord$partial-order type-hierarchy)
    (ord$partial-order concept-hierarchy)
    (ord$partial-order subtype)
    (= concept-hierarchy type-hierarchy)
    (= subtype concept-hierarchy)
    (= (ord$set type-hierarchy) type-label)
```

○ The type hierarchy contains two primitive type labels: the universal type '`entity`' and the absurd type '`absurdity`'. The symbol $\top$ is synonymous with '`entity`', and the symbol $\bot$ is synonymous with '`absurdity`'.

```
(3) (type-label entity)
    (((ord$upper-bound type-hierarchy) type-label) entity)
```

```
(4) (type-label absurdity)
    (((ord$lower-bound type-hierarchy) type-label) absurdity)
```

○ The *catalog of individuals* contains surrogates for actual individuals. They are the referents (individual markers) that appear in expressions (concepts) in the knowledge base.

```
(5) (set$set individual-marker)
```

○ The *relation* type *hierarchy* is a partially ordered set whose elements are called *relation labels*. Each relation label is specified as primitive or defined, but not both. We ignore defined relation labels at this point – defining relations with CG lambda expressions corresponds to specifying IFF type language interpretations. For the present, we assume that all relation labels are primitive.

```
(6) (set$set primitive)
```

```
(7) (set$set defined)
    (set$disjoint primitive defined)
```

```
(8) (set$set relation-label)
```

```
    (= relation-label (set$binary-union [primitive defined]))

(9) (ord$partial-order relation-hierarchy)
    (= (ord$set relation-hierarchy) relation-label)
```

- For every relation label, there is a natural number *n* called its *valence*. For every *n*-adic conceptual relation *r*, there is a sequence of *n* concept types, called the signature of *r*. All conceptual relations of the same relation type ρ have the same valence and the same signature. We lift these to the type ρ. Then, all relational instances have the same valence, arity and signature as any of their relational types.

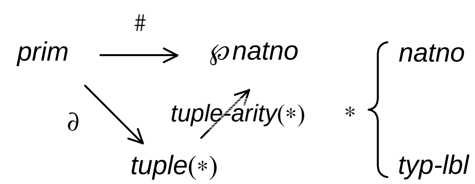
  

  Figure 1: Hypergraph Representation for Conceptual Graphs

  Every conceptual relation *r* has a relation type ρ = *sign*(*r*) and a nonnegative integer *n* = *val*(*r*) = |*arity*(*r*)| called its valence. A conceptual relation of valence *n* is said to be *n*-adic, and its arcs are numbered from 0 to *n*–1. For every *n*-adic conceptual relation *r*, there is a sequence (tuple) of *n* concept types $t_0, \ldots, t_{n-1}$ called the *signature* of *r*. The IFF Model Ontology represents a signature as a map from an arity set to a type set. The type set in question here is the concept type labels *typ-lbl*. In general, the arity set is a subset of a set of names. The simplest model for this situation is to let the first n natural number *arity*(*r*) = {0, … , *n*–1} ⊆ *natno* be the arity set with the set of natural numbers themselves as the set of names.

```
(10) (set.ftn$function valence)
     (= (set.ftn$source valence) primitive)
     (= (set.ftn$target valence) set$natno)

(11) (set.pr$pair pair)
     (= (set.pr$set1 pair) set$natno)
     (= (set.pr$set2 pair) type-label)

(12) (set.ftn$function arity)
     (= (set.ftn$source arity) primitive)
     (= (set.ftn$target arity) (set$power set$natno))
     (= arity (set.ftn$composition [valence (ord$down-embedding ord$natno)]))

(13) (set.ftn$function signature)
     (= (set.ftn$source signature) primitive)
     (= (set.ftn$target signature) (set.pr$tuple pair))
     (= signature (set.ftn$composition [arity (set.pr$tuple-assign pair)]))
```

- Here is an example of some ontologically-oriented assertions taken from the [Conceptual Graphs Standard](), which represent the population of the entity and relation type hierarchies. These declare entity and relation types, and to constrain these through subtyping and disjointness relationships.

```
(type-label person)
(type-label object)
(type-label cat)
(type-label mat)
(type-label action)
(type-label city)
(type-label bus)
(type-label on)

(subtype cat object)
(subtype mat object)
(subtype person object)
(subtype city object)
(subtype bus object)
(subtype on action)

(primitive on)
(= (valence on) 2)
(= ((signature on) 0) object)
(= ((signature on) 1) object)
(primitive agnt)
```

```
(= (valence agnt) 2)
(= ((signature agnt) 0) action)
(= ((signature agnt) 1) object)
(primitive dest)
(= (valence dest) 2)
(= ((signature dest) 0) action)
(= ((signature dest) 1) object)
(primitive inst)
(= (valence inst) 2)
(= ((signature inst) 0) action)
(= ((signature inst) 1) object)
```

- The hypergraph (Figure 1) for conceptual graphs is defined using these components.

```
(14) (hgph$hypergraph hypergraph)
     (= (hgph$name hypergraph) set$natno)
     (= (hgph$node hypergraph) type-label)
     (= (hgph$edge hypergraph) primitive)
     (= (hgph$reference hypergraph) pair)
     (= (hgph$tuple hypergraph) signature)
     (= (hgph$edge-arity hypergraph) arity)
```

Spangraphs

`cg.sgph`

In this section we discuss how conceptual graphs define spangraphs. We do this by functorially transforming hypergraphs to spangraphs. This spangraph representaton corresponds closely to the display form for conceptual graphs.

$$\text{Hypergraph} \;\Rightarrow\; \text{Spangraph}$$

○ We can complete this in one full swoop by using the spangraph function on hypergraphs.

```
(1) (sgph$spangraph spangraph)
    (= spangraph (hgph$spangraph cg$hypergraph))
```

○ But we expand on this somewhat, thereby getting more useful terminology.

– The set of *cases*

$$case = \sum arity = \sum_{\rho \in rel\text{-}lbl} arity(\rho) = \{(\rho, j) \mid \rho \in prim, j \leq val(\rho)\}$$

is the coproduct of its arity (Diagram 1).

– For any primitive relation label $\rho \in prim$ the case *injection* function

$$inj(\rho) : \#(\rho) = arity(\rho) \rightarrow case$$

is defined by $inj(\rho)(j) = (\rho, j)$ for all primitive relation labels $\rho \in prim$ and all natural numbers $j \leq val(\rho)$.

$$\begin{tikzcd} arity(\rho) \arrow[r, "inj(\rho)"] \arrow[dr, phantom, "\subseteq"] & case \arrow[d, "proj"] \\ & natno \end{tikzcd}$$

Diagram 1: Case and Injection

– The *indication* and *projection* functions (Diagram 2)

$$indic : case \rightarrow prim \text{ and}$$

$$proj : case \rightarrow natno,$$

which are based on the coproduct arity, are defined by

$$indic((\rho, j)) = \rho \text{ and } proj(\boldsymbol{G})((\rho, j)) = j$$

for all primitive relation labels $\rho \in prim$ and all natural numbers $j \leq val(\rho)$.

$$\begin{tikzcd} & case \arrow[dl, "indic"'] \arrow[dr, "proj"] & \\ prim \arrow[rr, "\#"] & & \wp natno \end{tikzcd} \quad natno$$

Diagram 2: Indication and Projection

– The *comediator* function (Diagram 3)

$$\tilde{*} = comed : case \rightarrow typ\text{-}lbl.$$

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

$$comed((\rho, j)) = \partial(\rho)(j)$$

for all primitive relation labels $\rho \in prim$ and all natural numbers $j \leq val(\rho)$.

$$case \xrightarrow{comed} typ\text{-}lbl$$

Diagram 3: Comediator

```
(2) (set$set case)
    (= case (hgph$case cg$hypergraph))

(3) (SET.FTN$function injection)
    (= (SET.FTN$source injection) cg$primitive)
    (= (SET.FTN$target injection) set.ftn$function)
    (= injection (hgph$injection cg$hypergraph))

(4) (set.ftn$function indication)
    (= (set.ftn$source indication) case)
    (= (set.ftn$target indication) cg$primitive)
    (= indication (hgph$indication cg$hypergraph))

(5) (set.ftn$function projection)
    (= (set.ftn$source projection) case)
    (= (set.ftn$target projection) set$natno)
    (= projection (hgph$projection cg$hypergraph))

(6) (set.ftn$function comediator)
    (= (set.ftn$source comediator) case)
    (= (set.ftn$target comediator) cg$type-label)
```

```
(= comediator (hgph$comediator cg$hypergraph))
```

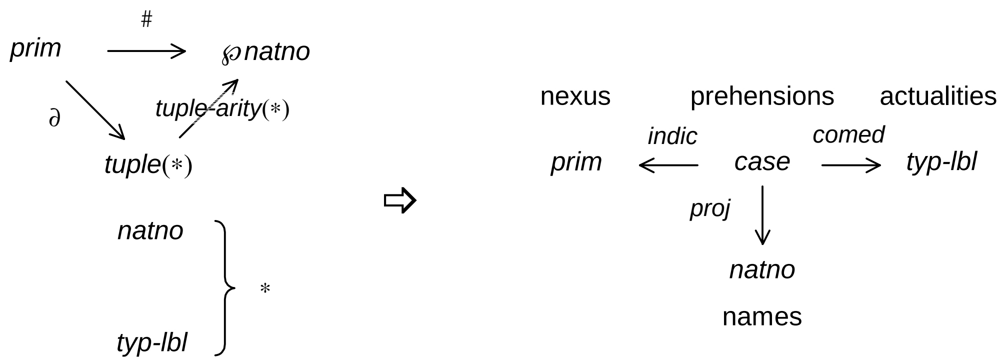


Figure 2: Passage to the Spangraph Representation for Conceptual Graphs

- Collecting these components together gives (Figure 2) a spangraph representation for conceptual graphs.

```
(7) (sgph$spangraph spangraph)
    (= (sgph$vertex spangraph) case)
    (= (sgph$first spangraph) comediator)
    (= (sgph$set1 spangraph) cg$type-label)
    (= (sgph$second spangraph) projection)
    (= (sgph$set2 spangraph) set$natno)
    (= (sgph$third spangraph) indication)
    (= (sgph$set3 spangraph) cg$primitive)
    (= spangraph (hgph$spangraph cg$hypergraph))
```

- We define two component functions for type languages.
  - The *arity* function (Diagram 4)

    $\#_1 = \mathit{arity} : \mathit{prim} \rightarrow \wp\mathit{case}$

    which is the fiber of the indication function, is defined by

    $\mathit{arity}(\rho) = \{(\rho, j) \mid j \leq \mathit{val}(\rho)\}$

    for all primitive relation labels $\rho \in \mathit{prim}$. The arity function commutes with the edge arity function.

$$\begin{tikzcd} \mathit{prim} \arrow[r, "\#_1"] \arrow[dr, "\#_0"'] & \wp\mathit{case} \arrow[d, "\mathit{proj}"] \\ & \wp\mathit{natno} \end{tikzcd}$$

Diagram 4: Arity

  - The *signature* function (Diagram 5)

    $\partial_1 = \mathit{sign} : \mathit{prim} \rightarrow \mathit{sign}(\tilde{*})$

    which is the composition of the arity function with the signature-assign of the comediator function, is defined by

    $\mathit{sign}(\rho) = \{t_0, \ldots, t_{n-1}\}$

    for all primitive relation labels $\rho \in \mathit{prim}$; that is, $\mathit{sign}(\rho)(j) = t_j$ for all primitive relation labels $\rho \in \mathit{prim}$ and all natural numbers $j \leq \mathit{val}(\rho)$.

$$\begin{tikzcd} \mathit{prim} \arrow[rr, "\#_1"] \arrow[dr, "\partial_1"'] & & \wp\mathit{case} \\ & \mathit{sign}(\tau) \arrow[ur, "\mathit{type\text{-}arity}"'] & \end{tikzcd}$$

Diagram 5: Signature

```
(8) (set.ftn$function arity)
    (= (set.ftn$source arity) cg$primitive)
    (= (set.ftn$target arity) (set$power case))
    (= arity (set.ftn$fiber indication))

(9) (set.ftn$function signature)
    (= (set.ftn$source signature) cg$primitive)
    (= (set.ftn$target signature) (set.ftn$signature comediator))
    (= signature (set.ftn$composition [arity (set.ftn$signature-assign comediator)]))
```

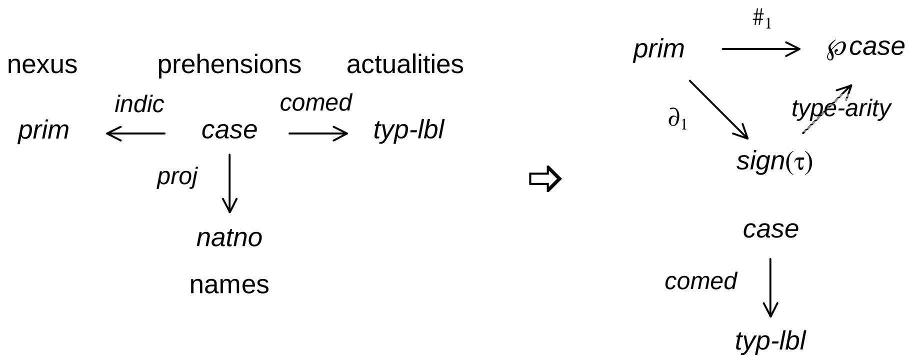


Figure 3: Passage to a Type Language Representation for Conceptual Graphs

- A type language representation for conceptual graphs is obtained (Figure 3) by applying the language functor to the spangraph. The details of this representation involve the arity and signature functions defined above.

```
(10) (lang$language language)
     (= (lang$variable language) case)
     (= (lang$entity language) cg$type-label)
     (= (lang$relation language) cg$primitive)
     (= (lang$reference language) comediator)
     (= (lang$signature language) signature)
     (= (lang$relation-arity language) arity)
     (= language (sgph$language spangraph))
```

Languages

`cg.lang`

As is clear from the definition, the variables for the type languages defined via spangraphs are restricted to elements of the case set; they are only of the form (ρ, j) for relation type (relation label) $\rho \in$ *rel-lbl* and natural number $j \leq$ *val*(ρ). These elements can be identified with the arcs for the display form of conceptual graphs. In order to define complex expressions (conceptual graphs), these variables are not flexible enough. For this the display form uses coreference links, whereas the linear form and the KIF and IFF representations of conceptual graphs use (sorted) logical variables.

- There is a set of *variables* (coreference labels), which includes the set of cases, since these are used for "variables" in relation type arities and signatures in the CG spangraph namespace. The set of variables is sorted (partitioned) by a *type* function according to entity type. There is a *type* function (Diagram 6)

  $\tau =$ *type* : *var* $\rightarrow$ *typ-lbl*

  which maps variables to entity types (type labels). For coherence, the type function cannot be defined on just natural numbers (thought of as variables in the CG hypergraph), since in general two different relation types (relation labels) may assign different entity types (type labels) to a particular natural number common to their arities. For this reason we include the set of cases, not the set of natural numbers, in the set of variables: case $\subseteq$ var. The type function extends the comediator function, which is defined on cases. We reiterate that the type function represents conceptual graphs in terms of <u>many-sorted</u> 1^{st}-order logic/language.

$$\begin{tikzcd} \mathit{case} \arrow[r] \arrow[dr, "\mathit{comed}"'] & \mathit{var} \arrow[d, "\tau = \mathit{type}"] \\ & \mathit{typ\text{-}lbl} \end{tikzcd}$$

Diagram 6: Type

```
(1) (set$set variable)
    (set$set coreference-set)
    (= variable coreference-set)
    (set$subset cg.sgph$case variable)

(2) (set.ftn$function type)
    (= (set.ftn$source type) variable)
    (= (set.ftn$target type) cg$type-label)
    (= comediator
       (set.ftn$composition [(set.ftn$inclusion [cg.sgph$case variable]) type]))
```

- There is a *primitive arity* function (Diagram 7)

  $\# =$ *prim-arity* : *prim* $\rightarrow \wp$*var*

  which extends the arity function in the CG spangraph namespace. So, pointwise this is defined by

  prim-arity$(\rho) = \{(\rho, j) \mid j \leq$ *val*$(\rho)\}$

  for all primitive relation labels $\rho \in$ *prim*. It is abstractly defined as the composition in Diagram 7.

$$\begin{tikzcd} & \wp\mathit{var} \\ \mathit{prim} \arrow[ur, "\#"] \arrow[r, "\#_1"'] & \wp\mathit{case} \arrow[u, "\wp\mathit{inc}"'] \end{tikzcd}$$

Diagram 7: Primitive Arity

  There is a *primitive signature* function (Diagram 8)

  $\partial =$ *sign* : *prim* $\rightarrow$ *sign*(τ)

  which is intuitively defined by

  sign$(\rho) = \{t_0, \ldots, t_{n-1}\}$

  for all primitive relation labels $\rho \in$ *prim*; that is, *sign*$(\rho)(j) = t_j$ for all primitive relation labels $\rho \in$ *prim* and all natural numbers $j \leq$ *val*(ρ). It is abstractly defined (Diagram 8) to be the composition of the primitive arity function with the signature-assign of the type function.

$$\begin{tikzcd} \mathit{prim} \arrow[r, "\#"] \arrow[dr, "\partial"'] & \wp\mathit{var} \arrow[d, shift left, "\mathit{type\text{-}assign}"] \\ & \mathit{sign}(\tau) \arrow[u, shift left, "\mathit{type\text{-}arity}"] \end{tikzcd}$$

Diagram 8: Primitive Signature

```
(3) (set.ftn$function primitive-arity)
    (= (set.ftn$source primitive-arity) cg$primitive)
    (= (set.ftn$target primitive-arity) (set$power variable))
    (= primitive-arity
       (set.ftn$composition
```

```
            [cg.sgph$arity (set.ftn$power (set.ftn$inclusion [cg.sgph$case variable]))]))

    (4) (set.ftn$function primitive-signature)
        (= (set.ftn$source primitive-signature) cg$primitive)
        (= (set.ftn$target primitive-signature) (set.ftn$signature type))
        (= primitive-signature
           (set.ftn$composition [primitive-arity (set.ftn$signature-assign type)]))
```

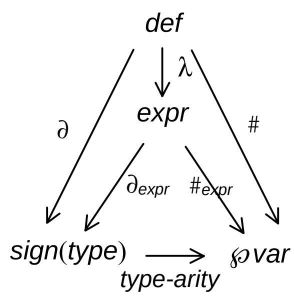


Diagram 9: Lambda

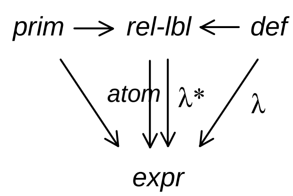


Diagram 10: Lambda-star

- For any nonnegative integer n, an n-adic *lambda expression* φ is a conceptual graph, called the body of φ, in which n concepts have been designated as formal parameters of φ. The formal parameters of φ are numbered from 0 to n–1. There is a sequence (tuple) of n concept types $(t_0, \ldots, t_{n-1})$ called the *signature* of φ, where each t_j is the concept type of the j-th formal parameter of φ. In the IFF lambda expressions have the same representation as conceptual graphs – they are represented as expressions.
- As an example consider the sentence ***"John is going to Boston."***
  - The conceptual graph for could be converted to the following dyadic lambda expression in linear form by replacing the name John with the symbol λ_0 and the name Boston with λ_1.

    ```
    [Person: λ0]←(Agnt)←[Go]→(Dest)→[City: λ1]
    ```

  - To simplify the parsing, the CGIF notation avoids the character λ and represents lambda expressions in a form that shows the signature explicitly.

    ```
    (lambda (Person*x, City*y) [Go *z] (Agnt ?z ?x) (Dest ?z ?y))
    ```

  - More detailed is the following IFF expression for this. Let φ represent the open KIF expression

    ```
    (exists (?z (go ?z))
        (and (agnt ?z ?x) (dest ?z ?y)))
    ```

    with *val* = 2, *arity* = $\{x, y\}$ (having the two free variables x and y), and signature *sign*(φ)(x) = Person and *sign*(φ)(y) = City. Substitutions are used as before.

    ```
    (cg.expr$expression agnt-expr0)
    (= agnt-expr0 (cg.expr$atom agnt))

    (lang$substitution agnt-subst)
    (= (lang$domain agnt-subst) (cg.expr$arity agnt))
    (= (agnt-subst [agnt 0]) z)
    (= (agnt-subst [agnt 1]) x)

    (cg.expr$expression agnt-expr)
    (= agnt-expr (substitution [agnt-expr0 agnt-subst]))
    ```

    …

    ```
    (cg.expr$expression conjoin)
    (= conjoin0 (cg.expr$conjunction [agnt-expr dest-expr]))

    (cg.expr$expression phi)
    ```

```
(= phi (cg.expr$existential-quantification [conjoin z]))

((cg.expr$arity phi) x)
((cg.expr$arity phi) y)
(= ((cg.expr$signature phi) x) person)
(= ((cg.expr$signature phi) y) city)
```

- For every defined relation label of valence *n* (definiendum), there is exactly one *n*-adic lambda expression (definiens), called its definition. There is a definition (interpretation) function (Diagram 9)

  – $lambda : def \rightarrow expr$.

  The arity and signature of a defined relation label is the arity and signature of its definiens.

  There is a *defined arity* function

  $\# = def\text{-}arity : def \rightarrow \wp var$

  which is abstractly defined (Diagram 1) to be the composition of lambda with the expression arity function.

  There is a *defined signature* function

  $\partial = def\text{-}sign : def \rightarrow sign(\tau)$

  which is abstractly defined (Diagram 9) to be the composition of lambda with the expression signature.

```
(5) (set.ftn$function lambda)
    (= (set.ftn$source lambda) cg$defined)
    (= (set.ftn$target lambda) cg.expr$expression)

(6) (set.ftn$function defined-arity)
    (= (set.ftn$source defined-arity) cg$defined)
    (= (set.ftn$target defined-arity) (set$power variable))
    (= defined-arity (set.ftn$composition [lambda cg.expr$arity]))

(7) (set.ftn$function defined-signature)
    (= (set.ftn$source defined-signature) cg$defined)
    (= (set.ftn$target defined-signature) (set.ftn$signature type))
    (= defined-signature (set.ftn$composition [lambda cg.expr$signature]))
```

$$\begin{tikzcd} prim \arrow[r] \arrow[dr, "\#_{prim}"'] & rel\text{-}lbl \arrow[d, "\#"] & def \arrow[l] \arrow[dl, "\#_{def}"] \\ & \wp var & \end{tikzcd}$$

Diagram 11: Arity

$$\begin{tikzcd} prim \arrow[r] \arrow[dr, "\partial_{prim}"'] & rel\text{-}lbl \arrow[d, "\partial"] & def \arrow[l] \arrow[dl, "\partial_{def}"] \\ & sign(\tau) & \end{tikzcd}$$

Diagram 12: Signature

- There is a *relation arity* function (Diagram 11)

  $\# = arity : rel\text{-}lbl \rightarrow \wp var$

  which combines the primitive and defined arity functions.

  There is a *relation signature* function (Diagram 12)

  $\partial = sign : rel\text{-}lbl \rightarrow sign(\tau)$

  which combines the primitive and defined signature functions.

```
(8) (set.ftn$function arity)
    (= (set.ftn$source arity) cg$relation-label)
    (= (set.ftn$target arity) (set$power variable))
    (forall (?rho (relation-label ?rho))
        (and (=> (cg$primitive ?rho)
                 (= (arity ?rho) (primitive-arity ?rho)))
             (=> (cg$defined ?rho)
                 (= (arity ?rho) (defined-arity ?rho)))))
```

```
(9) (set.ftn$function signature)
    (= (set.ftn$source signature) cg$relation-label)
    (= (set.ftn$target signature) (set.ftn$signature type))
    (forall (?rho (relation-label ?rho))
        (and (=> (cg$primitive ?rho)
                 (= (signature ?rho) (primitive-signature ?rho)))
             (=> (cg$defined ?rho)
                 (= (signature ?rho) (defined-signature ?rho)))))
```

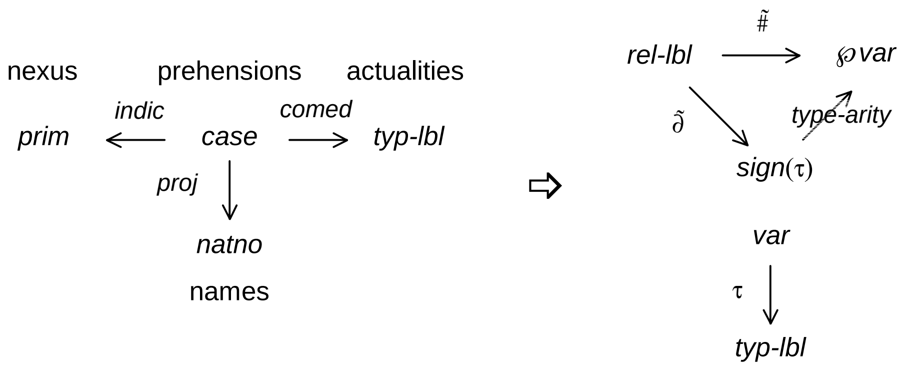


Figure 4: Passage to a Type Language Representation for Conceptual Graphs

- An extended type language representation for conceptual graphs is obtained (Figure 4) by collecting together the components above.

```
(10) (lang$language language)
     (= (lang$variable language) variable)
     (= (lang$entity language) cg$type-label)
     (= (lang$relation language) cg$relation-label)
     (= (lang$reference language) type)
     (= (lang$signature language) signature)
     (= (lang$relation-arity language) arity)
```

Expressions

cg.expr

- The set of all expressions (conceptual graphs) definable in a knowledge base forms a type language that extends the basic type language of the knowledge base.

```
(1) (set$set expression)
    (= expression (lang.expr$set cg.lang$language))

(2) (set.ftn$function arity)
    (= (set.ftn$source arity) expression)
    (= (set.ftn$target arity) (set$power cg.lang$variable))
    (= arity (lang.expr$arity cg.lang$langauge))

(3) (set.ftn$function signature)
    (= (set.ftn$source signature) expression)
    (= (set.ftn$target signature) (set.ftn$signature type))
    (= signature (lang.expr$signature cg.lang$langauge))

(4) (lang$language expression-language)
    (= expression-language (lang.expr$expression cg.lang$language))

    (= (lang$variable expression-language) cg.lang$variable)
    (= (lang$entity expression-language) cg$type-label)
    (= (lang$reference expression-language) cg.lang$reference)
    (= (lang$relation expression-language) expression)
    (= (lang$relation-arity expression-language) arity)
    (= (lang$signature expression-language) signature)
```

- The set of all expressions (conceptual graphs) definable in a knowledge base forms a classification.

```
(5) (cls$classification expression-classification)
    (= (cls$instance expression-classification) cg.mod$tuple)
    (= (cls$type expression-classification) expression)
```

- Expressions (conceptual graphs) can be built in a recursive fashion by using the following operators.

```
(6) (set$set expression-pair)
    (= expression-pair
       (set.lim.prd2$binary-product [expression expression]))

(7) (set$set expression-variable-pair)
    (= expression-variable-pair
       (lang$case cg.lang$language))

(8) (set$set expression-substitution-pair)
    (= (expression-substitution-pair
       (rel$extent (lang$substitutible cg.lang$language)))

(9) (set.ftn$function atom)
    (= (set.ftn$source atom) cg$relation-label)
    (= (set.ftn$target atom) expression)
    (= atom (lang.expr$atom cg.lang$language))

(10) (set.ftn$function negation)
     (= (set.ftn$source negation) expression)
     (= (set.ftn$target negation) expression)
     (= negation (lang.expr$negation cg.lang$language))

(11) (set.ftn$function conjunction)
     (= (set.ftn$source conjunction) expression-pair)
     (= (set.ftn$target conjunction) expression)
     (= conjunction (lang.expr$conjunction cg.lang$language))

(12) (set.ftn$function disjunction)
     (= (set.ftn$source disjunction) expression-pair)
     (= (set.ftn$target disjunction) expression)
     (= disjunction (lang.expr$disjunction cg.lang$language))

(13) (set.ftn$function implication)
```

```
      (= (set.ftn$source implication) expression-pair)
      (= (set.ftn$target implication) expression)
      (= implication (lang.expr$implication cg.lang$language))

(14)  (set.ftn$function equivalence)
      (= (set.ftn$source equivalence) expression-pair)
      (= (set.ftn$target equivalence) expression)
      (= equivalence (lang.expr$equivalence cg.lang$language))

(15)  (set.ftn$function existential-quantification)
      (= (set.ftn$source existential-quantification) expression-variable-pair)
      (= (set.ftn$target existential-quantification) expression)
      (= existential-quantification
         (lang.expr$existential-quantification cg.lang$language))

(16)  (set.ftn$function universal-quantification)
      (= (set.ftn$source universal-quantification) expression-variable-pair)
      (= (set.ftn$target universal-quantification) expression)
      (= universal-quantification
         (lang.expr$universal-quantification cg.lang$language))

(17)  (set.ftn$function substitution)
      (= (set.ftn$source substitution) expression-substitution-pair)
      (= (set.ftn$target substitution) expression)
      (= substitution (lang.expr$substitution cg.lang$language))
```

- Here is an example of how these operations can be used in a sntax-directed build for the IFF representation of a conceptual graph. Consider the natural language assertion ***"John is going to Boston by bus"*** taken from the Conceptual Graphs Standard. This can be represented by the following *simple* conceptual graph.

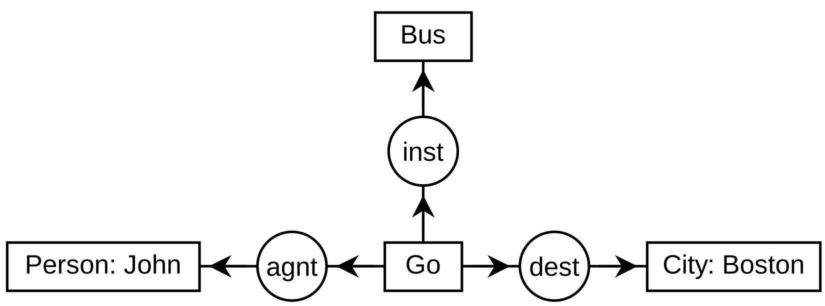


Figure 5: Simple Conceptual Graph

  – Figure 5 represents this in display form.
  – The linear form for this is the following expression.

```
[Go]-
   (Agnt)→[Person: John]
   (Dest)→[City: Boston]
   (Inst)→[Bus]
```

  – An canonical interchange version is the following expression.

```
[Go *x] [Bus *w] [Person: John] [City: Boston]
(Agnt ?x ?y) (Dest ?x ?z) (Inst ?x ?w)
```

  – This can be expressed in KIF notation as follows. The first two assertions are entity classifications. The last assertion is an expression classification.

```
(person john)
(city boston)

(exists (?x (go ?x) ?w (bus ?w))
    (and (agnt ?x john) (dest ?x boston) (inst ?x ?w)))
```

– More detailed is the following IFF expression for this. Let φ represent the open KIF expression

```
(exists (?x (go ?x) ?w (bus ?w))
    (and (agnt ?x ?y) (dest ?x ?z) (inst ?x ?w)))
```

with *val* = 2, *arity* = {*y*, *z*} (having the two free variables *y* and *z*). Then the complete expression represents the following incidences. The first two incidences in an entity classification; that is, in the catelogue of individuals. The final incidence is in the expression classification.

John ⊨ Person

Boston ⊨ City

(John, Boston) ⊨ φ

Implicit in the transformation from display form to IFF is the introduction of variables. This introduction corresponds to the application of a substitution to the various primitive expressions. For example, the agent relation type, regarded as a primitive expression, has (relational) arity {0, 1}. Looking forward, we want to replace (the natural number regarded as) the variable {agnt, 0} with the variable *x* and we want to replace the variable {agnt, 1} with the variable *y*. The application of the atom, conjunction, … operators is syntax-directed, and would be accomplished during a parse of the CGIF expression.

```
(expression agnt-expr0)
(= agnt-expr0 (atom agnt))

(lang$substitution agnt-subst)
(= (lang$domain agnt-subst) (arity agnt))
(= (agnt-subst [agnt 0]) x)
(= (agnt-subst [agnt 1]) y)

(expression agnt-expr)
(= agnt-expr (substitution [agnt-expr0 agnt-subst]))
```

…

```
(expression conjoin0)
(= conjoin0 (conjunction [agnt-expr dest-expr]))

(expression conjoin)
(= conjoin (conjunction [conjoin0 inst-expr]))

(expression phi0)
(= phi0 (existential-quantification [conjoin w]))

(expression phi)
(= phi (existential-quantification [phi0 x]))

(= (valence phi) 2)
((arity phi) y)
((arity phi) z)
(= ((signature phi) y) object)
(= ((signature phi) z) object)

(tuple pair)
(= (tuple-length pair) 2)
((tuple-arity pair) y)
((tuple-arity pair) z)
(= ((tuple-signature pair) y) john)
(= ((tuple-signature pair) z) boston)

(expression-classification [pair phi])
```

Interpretations

cg.interp

- The definition function can be extended to all relation symbols as the function (Diagram 10)
  - *lambda** : *rel-lbl* → *expr*

  which is the copairing of *lamba* with *atom*, the primitive relation type injection.

```
(1) (set.ftn$function lambda-star)
    (= (set.ftn$source lambda-star) cg$relation-label)
    (= (set.ftn$target lambda-star) cg.expr$expression)
    (forall (?rho (relation-label ?rho))
        (and (=> (cg$primitive ?rho)
                 (= (lambda-star ?rho) (cg.expr$atom ?rho)))
             (=> (cg$defined ?rho)
                 (= (lambda-star ?rho) (lambda ?rho)))))
```

- This definition (interpretation) of relation symbols has an associated type interpretation
  - *interp* : *lang* → *expr*(*lang*),

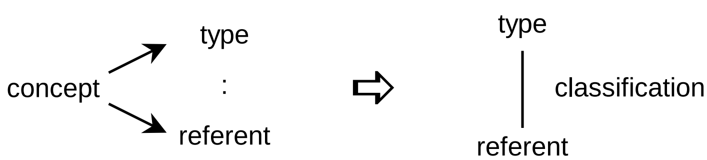
  whose variable and entity functions are identity, and whose relation function is lambda-star.

```
(2) (lang.interp$interpretation interpretation)
    (= (lang.interp$source interpretation) cg.lang$language)
    (= (lang.interp$target interpretation) cg.expr$expression-language)
    (= (lang.mor$variable (lang.interp$morphism interpretation))
       (set.ftn$identity cg.lang$variable))
    (= (lang.mor$entity (lang.interp$morphism interpretation))
       (set.ftn$identity cg$type-label))
    (= (lang.mor$relation (lang.interp$morphism interpretation)) lambda-star)
```

Models

This section describes the IFF representation for a knowledge base as described in the Conceptual Graphs Standard. The type language for these knowledgebases was defined in the CG type language namespace. It corresponds closely to the interchange form of conceptual graphs. A knowledge base consists of the following components.

- The entity instances (individual markers) and entity types (type labels) form a classification. Incidence in this classification is specified in the concepts in the knowledge base by the link between referents

  and type labels. Every concept has a *concept type* t and a *referent r*. Incidence is represented by the colon separator between type label and referent in a concept.

```
(1) (cls$classification entity-classification)
    (= (cls$instance entity-classification) universe)
    (= (cls$type entity-classification) cg$type-label)
```

- Here is an example of how the singleton graphs in the catalog of individuals are represented in IFF in terms of the entity classification. Consider the natural language assertion ***"John is a person."*** This can be represented by a *singleton* conceptual graph – it consists of a single concept, but no conceptual relations or arcs.

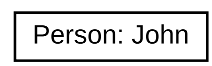
  

  Figure 6: Singleton Conceptual Graph

  – Figure 6 represents this in display form.

  – The linear form (and interchange form) for this is the following expression.

```
[Person: John]
```

  – This can be expressed in KIF notation as follows.

```
(person john)
```

  – This concept represents the following incidence in the entity classification of a knowledge base.

  John $\models$ Person

  – The IFF notation for this was listed above.

```
(cg.mod$individual-marker john)
(cg.mod$entity-classification [john person])
```

  Here is an example of some ontologically-oriented assertions taken from the Conceptual Graphs Standard, which represent the population of the catalogue of individuals.

```
(cg$individual-marker john)
(cg$individual-marker yojo)
(cg$individual-marker #2631)
(cg$individual-marker boston)

(cg$entity-classification [john person])
(cg$entity-classification [yojo cat])
(cg$entity-classification [#2631 mat])
(cg$entity-classification [boston city])
```

- The *tuples* (relation instances) are abstractions for ordinary tuples. They correspond to the entries in a relational database table. They appear implicitly as the arguments in the atomic expressions (conceptual relations) that appear in expressions (concepts) in the knowledge base.

```
(2) (set$set tuple)
```

- The relation instances (tuples) and relation types (relation labels) form a classification. Incidence in this classification is specified in the knowledge base as the conceptual relations (primitive relational expressions or star conceptual graphs) that resolve any expression (conceptual graph).

```
(3) (cls$classification relation-classification)
    (= (cls$instance relation-classification) tuple)
    (= (cls$type relation-classification) relation-label)
```

- A *knowledge base* is another name for a model. Every knowledge base (model) has an associated type language. This language is defined in the CG type language namespace. The contents of the knowledge base must satisfy the following constraints:
  - The entity types (type labels), which in any expression (conceptual graph) or type interpretation (lambda expression) in the knowledge base, must be specified in the entity type hierarchy.
  - The relation types (relation labels), which in any expression (conceptual graph) or type interpretation (lambda expression) in the knowledge base, must be specified in the relation type hierarchy.
  - The entity instances (individual markers), which in any expression (conceptual graph) or type interpretation (lambda expression) in the knowledge base, must be specified in the catalog of individuals.

```
(4) (mod$model knowledge-base)
    (= (mod$universe knowledge-base) individual-marker)
    (= (mod$tuple knowledge-base) tuple)
    (= (mod$entity knowledge-base) entity-classification)
    (= (mod$relation knowledge-base) relation-classification)
    (= (mod$type knowledge-base) cg.lang$language)
    (= (mod$expression-classification knowledge-base)
       cg.expr$expression-classification)
```