

# A Declarative Semantics for SNOMED CT Expression Constraints

March 9, 2015

## Contents

<b>1</b>	<b>Axiomatic Data Types</b>	<b>2</b>
1.1	Atomic Data Types . . . . .	2
1.2	Composite Data Types . . . . .	3
<b>2</b>	<b>The Substrate</b>	<b>3</b>
2.1	Substrate Components . . . . .	3
2.2	Substrate . . . . .	3
2.2.1	Strict and Permissive Substrates . . . . .	6
2.2.2	Strict Substrate . . . . .	6
2.2.3	Permissive Substrate . . . . .	6
<b>3</b>	<b>SCTIDS or Error Return</b>	<b>6</b>
<b>4</b>	<b>Interpretation of Expression Constraints</b>	<b>7</b>
4.1	expressionConstraint . . . . .	7
4.1.1	unrefinedExpressionConstraint . . . . .	7
4.1.2	refinedExpressionConstraint . . . . .	8
4.1.3	simpleExpressionConstraint . . . . .	8
4.1.4	compoundExpressionConstraint . . . . .	9
4.1.5	conjunctionExpressionConstraint . . . . .	10
4.1.6	disjunctionExpressionConstraint . . . . .	10
4.1.7	exclusionExpressionConstraint . . . . .	11
4.1.8	subExpressionConstraint . . . . .	11
4.2	refinement . . . . .	12
4.2.1	conjunctionGroup . . . . .	12
4.2.2	disjunctionGroup . . . . .	12
4.2.3	subRefinement . . . . .	13
4.3	attributeSet . . . . .	13
4.3.1	conjunctionAttributeSet . . . . .	14
4.3.2	disjunctionAttributeSet . . . . .	14
4.3.3	subAttributeSet . . . . .	15

4.4	attributeGroup . . . . .	15
4.5	attribute . . . . .	15
4.5.1	expressionAttribute . . . . .	16
4.5.2	concreteAttribute . . . . .	16
4.6	Group Cardinality . . . . .	17
4.7	Cardinality . . . . .	18
<b>5</b>	<b>Substrate Interpretations</b>	<b>19</b>
5.1	attributeName . . . . .	19
5.2	attributeExpressionConstraint . . . . .	19
5.3	concreteAttributeConstraint . . . . .	20
5.4	FocusConcept . . . . .	21
5.4.1	focusConcept . . . . .	21
5.4.2	memberOf . . . . .	22
5.5	ConceptReferences . . . . .	22
5.5.1	conceptId . . . . .	22
5.5.2	conceptReference . . . . .	23
<b>6</b>	<b>Glue and Helper Functions</b>	<b>23</b>
6.1	Types . . . . .	23
6.2	Result transformations . . . . .	23
<b>7</b>	<b>Appendix 1</b>	<b>26</b>
<b>8</b>	<b>Appendix 2</b>	<b>26</b>

# 1 Axiomatic Data Types

## 1.1 Atomic Data Types

This section identifies the atomic data types that are assumed for the rest of this specification, specifically:

- **SCTID** – a SNOMED CT identifier
- **TERM** – a fully specified name, preferred term or synonym for a SNOMED CT Concept
- **REAL** – a real number
- **STRING** – a string literal
- **GROUP** – a role group identifier
- $\mathbb{N}$  – a non-negative integer
- $\mathbb{Z}$  – an integer

$[SCTID, TERM, REAL, STRING, GROUP]$

We will also need to recognize some well known identifiers: the *is\_a* attribute, the *zero\_group* and *attribute\_concept*, the parent of all attributes

$is\_a : SCTID$ $zero\_group : GROUP$ $attribute\_concept : SCTID$ $refset\_concept : SCTID$	
---	--

## 1.2 Composite Data Types

While we can't fully specify the behavior of the concrete data types portion of the specification at this point, it is still useful to spell out the anticipated behavior on an abstract level.

- **CONCRETEVALUE** – a string, integer or real literal
- **TARGET** – the target of a relationship that is either an *SCTID* or a *CONCRETEVALUE*

$$CONCRETEVALUE ::= string\langle\langle STRING \rangle\rangle \mid integer\langle\langle \mathbb{Z} \rangle\rangle \mid real\langle\langle REAL \rangle\rangle$$

$$TARGET ::= object\langle\langle SCTID \rangle\rangle \mid concrete\langle\langle CONCRETEVALUE \rangle\rangle$$

## 2 The Substrate

A substrate represents the context of an interpretation.

### 2.1 Substrate Components

**Quad** Relationships in the substrate are represented a 4 element tuples or “quads” which consist of a source, attribute, target and role group identifier. The *is\_a* attribute may only appear in the zero group, and the target of an *is\_a* attribute must be a *SCTID* (not a *CONCRETEVALUE*)

$Quad$ $s : SCTID$ $a : SCTID$ $t : TARGET$ $g : GROUP$	$a = is\_a \Rightarrow (g = zero\_group \wedge object \sim t \in SCTID)$
---	--

### 2.2 Substrate

A substrate consists of:

- **concepts** The set of *SCTID*s (concepts) that are considered valid in the context of the substrate. *References to any SCTID that is not a member of concepts MUST be treated as an error.*
- **relationships** A set of relationship quads (source, attribute, target, group)
- **parentsOf** A function from an *SCTID* to its asserted and inferred parents
- **equivalent\_concepts** A function from an *SCTID* to the set of *SCTID*'s that have been determined to be equivalent to it.

- **refsets** The reference sets within the context of the substrate whose members are members are concept identifiers (i.e. are in *concepts*). While not formally spelled out in this specification, it is assumed that the typical reference set function would be returning a subset of the *refsetId/referencedComponentId* tuples represented in one or more RF2 Refset Distribution tables.

The following functions can be computed from the basic set above

- **childrenOf** The inverse of the *parentsOf* function
- **descendants** The transitive closure of the *childrenOf* function
- **ancestors** The transitive closure of the *parentsOf* function
- **attributeIds** The descendantsOf the *attribute\_concept*, including equivalents
- **refsetsIds** The descendants of the *refset\_concept*, including equivalents

The formal definition of substrate follows, where *c* and *r* are given and the remainder are derived. The expressions below assert that:

1. All sources, attributes and SCTID targets of *relationships* are included in the substrate *concepts* list.
2. There is a *parentsOf* entry for every concept in the substrate *concepts* list.
3. Every *sctid* in the range of the *parentsOf* function is in the substrate *concepts* list.
4. Every *is\_a* relationship entry is represented in the *parentsOf* function. (Note that the reverse isn't necessarily true).
5. There is an *equivalent\_concepts* assertion for every substrate concept.
6. The *equivalent\_concepts* function is reflexive (i.e. every concept is equivalent to itself)
7. If two concepts (*c1* and *c2*) are equivalent, then they:
  - Have the same parents
  - Appear the subject, attribute and object of the same set of relationships
  - Appear in the domain of the same set of refsets
  - Both appear in the range of any refset that one appears in
8. Every refset is a substrate *concepts*
9. Every member of a refset is a substrate *concept*
10. *childrenOf* is the inverse of *parentsOf*, where any concept that isn't a parent has no children.
11. *descendants* is the transitive closure of the *childrenOf* function
12. *ancestors* is the transitive closure of the *parentsOf* function
13. No concept can be its own ancestor (or, by inference, descendant)
14. Every *attributeId* is a substrate *concept*
15. Every *refsetId* is a substrate *concept*

---

*Substrate*

---

$concepts : \mathbb{P} SCTID$

$relationships : \mathbb{P} Quad$

$parentsOf : SCTID \rightarrow \mathbb{P} SCTID$

$equivalent\_concepts : SCTID \rightarrow \mathbb{P} SCTID$

$refsets : SCTID \rightarrow \mathbb{P} SCTID$

$childrenOf : SCTID \rightarrow \mathbb{P} SCTID$

$descendants : SCTID \rightarrow \mathbb{P} SCTID$

$ancestors : SCTID \rightarrow \mathbb{P} SCTID$

$attributeIds : \mathbb{P} SCTID$

$refsetIdIds : \mathbb{P} SCTID$

---

$\forall rel : relationships \bullet rel.s \in concepts \wedge rel.a \in concepts \wedge$   
 $(object \sim rel.t \in SCTID \Rightarrow object \sim rel.t \in concepts)$

$dom\ parentsOf = concepts$

$\bigcup(ran\ parentsOf) \subseteq concepts$

$\forall r : relationships \bullet r.a = is\_a \Rightarrow (object \sim r.t) \in parentsOf\ r.a$

$dom\ equivalent\_concepts = concepts$

$\forall c : concepts \bullet c \in equivalent\_concepts\ c$

$\forall c1, c2 : concepts \mid c2 \in (equivalent\_concepts\ c1) \bullet$

$parentsOf\ c1 = parentsOf\ c2 \wedge$

$\{r : relationships \mid r.s = c1\} = \{r : relationships \mid r.s = c2\} \wedge$

$\{r : relationships \mid r.a = c1\} = \{r : relationships \mid r.a = c2\} \wedge$

$\{r : relationships \mid object \sim r.t = c1\} = \{r : relationships \mid object \sim r.t = c2\} \wedge$

$c1 \in dom\ refsets \Leftrightarrow c2 \in dom\ refsets \wedge$

$c1 \in dom\ refsets \Rightarrow refsets\ c1 = refsets\ c2 \wedge$

$(\forall rsd : ran\ refsets \bullet c1 \in rsd \Leftrightarrow c2 \in rsd)$

$dom\ refsets \subseteq concepts$

$\bigcup(ran\ refsets) \subseteq concepts$

$dom\ childrenOf = concepts$

$\forall s, t : concepts \bullet t \in parentsOf\ s \Leftrightarrow s \in childrenOf\ t$

$\forall c : concepts \mid c \notin \bigcup(ran\ childrenOf) \bullet childrenOf\ c = \emptyset$

$\forall s : concepts \bullet$

$descendants\ s = childrenOf\ s \cup \bigcup\{t : childrenOf\ s \bullet descendants\ t\}$

$\forall t : concepts \bullet$

$ancestors\ t = parentsOf\ t \cup \bigcup\{s : parentsOf\ t \bullet ancestors\ s\}$

$\forall t : concepts \bullet t \notin ancestors\ t$

$attributeIds \subseteq concepts$

$refsetIdIds \subseteq concepts$

---

### 2.2.1 Strict and Permissive Substrates

Implementations may choose to implement “strict” substrates, where additional rules apply or “permissive” substrates where rules are relaxed.

#### 2.2.2 Strict Substrate

A **strict\_substrate** is a substrate where:

- There is at least one *SCTID* that is not substrate concept
- Every *attributeId* must be a descendant of *attribute\_concept*
- Every *refsetId* must be a descendant of *refset\_concept*
- *relationship* attributes must be *attributeIds*
- *refset* domains must be *refsetIds*

<i>strict_substrate</i>	_____
<i>Substrate</i>	
<i>concepts</i>	$\subset SCTID$
<i>attributeIds</i>	$= descendants\ attribute\_concept$
$\forall r : relationships$	$\bullet r.a \in attributeIds$
<i>refsetIds</i>	$= descendants\ refset\_concept$
$dom\ refsets$	$\subseteq refsetIds$

#### 2.2.3 Permissive Substrate

A permissive substrate is a substrate where every query will return some result – all *SCTID*'s are considered valid.

This includes the following rules:

1. Every possible *SCTID* is a substrate concept, attribute and a valid refset
2. The refset function will return a (possibly empty) set of results for any refuted

<i>permissive_substrate</i>	_____
<i>Substrate</i>	
<i>concepts</i>	$= SCTID \wedge attributeIds = concepts \wedge refsetIds = concepts$

## 3 SCTIDS or Error Return

The result of applying a query against a substrate is either a (possibly empty) set of *SCTID*'s or an *ERROR*. An *ERROR* occurs when:

- The interpretation of a conceptId is not a substrate *concept*
- The interpretation of a relationship attribute is not a substrate *attributeId*
- The interpretation of a reset is not a substrate *refsetId*

$ERROR ::= unknownConceptReference \mid unknownAttributeId \mid unknownRefsetId$   
 $Sctids\_or\_Error ::= ok\langle\langle\mathbb{P}\ SCTID\rangle\rangle \mid error\langle\langle ERROR\rangle\rangle$

## 4 Interpretation of Expression Constraints

This section defines the interpretation of all language constructs that are interpreted in terms of other language constructs. Each interpretation that follows begins with a simplified version of the language construct in the specification. It then formally specifies the constructs that are used in the interpretation, followed by the interpretation itself. We start with the definition of *expressionConstraint*, which, once interpreted, returns either a set of SCTIDs or an error condition.

### 4.1 expressionConstraint

```
expressionConstraint = ws ( refinedExpressionConstraint / unrefinedExpressionConstraint )
ws
```

**expressionConstraint** takes either a **refinedExpressionConstraint** or **unrefinedExpressionConstraint** and returns its interpretation as either a set of SCTIDs or an error condition.

$$\begin{aligned} \text{expressionConstraint} ::= & \\ & \text{expcons\_refined} \langle\langle \text{refinedExpressionConstraint} \rangle\rangle \mid \\ & \text{expcons\_unrefined} \langle\langle \text{unrefinedExpressionConstraint} \rangle\rangle \end{aligned}$$

$i\_expressionConstraint : \\ Substrate \rightarrow expressionConstraint \rightarrow Sctids\_or\_Error$
$\forall ss : Substrate; ec : expressionConstraint \bullet i\_expressionConstraint ss ec =$
$\text{if } ec \in \text{ran expcons\_refined}$
$\quad \text{then } i\_refinedExpressionConstraint ss (expcons\_refined \sim ec)$
$\text{else } i\_unrefinedExpressionConstraint ss (expcons\_unrefined \sim ec)$

#### 4.1.1 unrefinedExpressionConstraint

The interpretation of an **unrefinedExpressionConstraint** is either the interpretation of a **compoundExpressionConstraint** or a **simpleExpressionConstraint**

```
unrefinedExpressionConstraint = compoundExpressionConstraint / simpleExpressionConstraint
```

$$\begin{aligned} \text{unrefinedExpressionConstraint} ::= & \\ & \text{unrefined\_compound} \langle\langle \text{compoundExpressionConstraint} \rangle\rangle \mid \\ & \text{unrefined\_simple} \langle\langle \text{simpleExpressionConstraint} \rangle\rangle \end{aligned}$$

$i\_unrefinedExpressionConstraint :$ $Substrate \rightarrow unrefinedExpressionConstraint \rightarrow Sctids\_or\_Error$
$\forall ss : Substrate; uec : unrefinedExpressionConstraint \bullet$ $i\_unrefinedExpressionConstraint ss uec =$ $\text{if } ucec \in \text{ran } unrefined\_compound$ $\text{ then } i\_compoundExpressionConstraint ss (unrefined\_compound \sim uec)$ $\text{ else } i\_simpleExpressionConstraint ss (unrefined\_simple \sim uec)$

#### 4.1.2 refinedExpressionConstraint

refinedExpressionConstraint = unrefinedExpressionConstraint ws ":" ws refinement / "(" ws  
refinedExpressionConstraint ws ")"

The interpretation of **refinedExpressionConstraint** is the intersection of the interpretation of the **unrefinedExpressionConstraint** and the **refinement**, both of which return a set of SCTID's or an error. The second production defines **refinedExpressionConstraint** in terms of itself and has no impact on the results.

$$refinedExpressionConstraint ==$$

$$unrefinedExpressionConstraint \times refinement$$

$i\_refinedExpressionConstraint :$ $Substrate \rightarrow refinedExpressionConstraint \rightarrow Sctids\_or\_Error$
$\forall ss : Substrate; rec : refinedExpressionConstraint \bullet$ $i\_refinedExpressionConstraint ss rec =$ $intersect (i\_unrefinedExpressionConstraint ss (first rec))(i\_refinement ss (second rec))$

$i\_refinedExpressionConstraint :$ $Substrate \rightarrow refinedExpressionConstraint \rightarrow Sctids\_or\_Error$
--

#### 4.1.3 simpleExpressionConstraint

The interpretation of **simpleExpressionConstraint** is the application of an optional constraint operator to the interpretation of **focusConcept**, which returns a set of SCTID's or an error. The interpretation of an error is the error.



```

simpleExpressionConstraint = [constraintOperator ws] focusConcept
bnfconstraintOperator = descendantOrSelfOf / descendantOf / ancestorOr-
SelfOf / ancestorOf

```

$simpleExpressionConstraint == constraintOperator[0..1] \times focusConcept$   
 $constraintOperator ::= descendantsOrSelfOf \mid descendantOf \mid ancestorOrSelfOf \mid ancestorOf$

```

i_simpleExpressionConstraint :
  Substrate → simpleExpressionConstraint → Sctids_or_Error

∀ ss : Substrate; sec : simpleExpressionConstraint •
  i_simpleExpressionConstraint ss sec =
    i_constraintOperator ss (first sec) (i_focusConcept ss (second sec))

```

#### 4.1.4 compoundExpressionConstraint

The interpretation of a *compoundExpressionConstraint* is the interpretation of its corresponding component.

```

compoundExpressionConstraint = conjunctionExpressionConstraint / disjunctionExpression-
Constraint / exclusionExpressionConstraint / "(" ws compoundExpressionConstraint ws ")"

```

$compoundExpressionConstraint ::=$   
 $compound\_conj \langle\langle conjunctionExpressionConstraint \rangle\rangle \mid$   
 $compound\_disj \langle\langle disjunctionExpressionConstraint \rangle\rangle \mid$   
 $compound\_excl \langle\langle exclusionExpressionConstraint \rangle\rangle$

```

i_compoundExpressionConstraint :
  Substrate → compoundExpressionConstraint → Sctids_or_Error

∀ ss : Substrate; cec : compoundExpressionConstraint •
  i_compoundExpressionConstraint ss cec =
    if cec ∈ ran compound_conj
    then i_conjunctionExpressionConstraint ss (compound_conj ~ cec)
    else if cec ∈ ran compound_disj
    then i_disjunctionExpressionConstraint ss (compound_disj ~ cec)
    else i_exclusionExpressionConstraint ss (compound_excl ~ cec)

```

The signature below is used because the definition of **compoundExpressionConstraint** is recursive

```

i_compoundExpressionConstraint' :
  Substrate → compoundExpressionConstraint → Sctids_or_Error

```

#### 4.1.5 conjunctionExpressionConstraint

`conjunctionExpressionConstraint` is interpreted the conjunction (intersection) of the interpretation of two or more `subExpressionConstraints`. The `conjunction` aspect is ignored because there is no other choice

```
conjunctionExpressionConstraint = subExpressionConstraint 1*(ws conjunction ws subExpressionConstraint)
```

$$\text{conjunctionExpressionConstraint} == \text{subExpressionConstraint} \times \text{seq}_1(\text{subExpressionConstraint})$$

Apply the intersection operator to the interpretation of each `subExpressionConstraint`

```
i_conjunctionExpressionConstraint :
  Substrate → conjunctionExpressionConstraint → Sctids_or_Error

∀ ss : Substrate; cecr : conjunctionExpressionConstraint •
  i_conjunctionExpressionConstraint ss cecr =
    applyToSequence ss i_subExpressionConstraint intersect cecr
```

#### 4.1.6 disjunctionExpressionConstraint

`disjunctionExpressionConstraint` is interpreted the disjunction (union) of the interpretation of two or more `subExpressionConstraints`. The `disjunction` element is ignored because there is no other choice.

```
disjunctionExpressionConstraint = subExpressionConstraint 1*(ws disjunction ws subExpressionConstraint)
```

$$\text{disjunctionExpressionConstraint} == \text{subExpressionConstraint} \times \text{seq}_1(\text{subExpressionConstraint})$$

Apply the union operator to the interpretation of each `subExpressionConstraint`

```
i_disjunctionExpressionConstraint :
  Substrate → disjunctionExpressionConstraint → Sctids_or_Error

∀ ss : Substrate; decr : disjunctionExpressionConstraint •
  i_disjunctionExpressionConstraint ss decr =
    applyToSequence ss i_subExpressionConstraint union decr
```

#### 4.1.7 exclusionExpressionConstraint

The interpretation `exclusionExpressionConstraint` removes the interpretation of the second `exclusionExpressionConstraint` from the interpretation of the first. Errors are propagated.

`exclusionExpressionConstraint = subExpressionConstraint ws exclusion ws subExpressionConstraint`

$$\text{exclusionExpressionConstraint} == \text{subExpressionConstraint} \times \text{subExpressionConstraint}$$

*i\_exclusionExpressionConstraint* :  
Substrate → exclusionExpressionConstraint → Sctids\_or\_Error

∀ ss : Substrate; ecr : exclusionExpressionConstraint •  
i\_exclusionExpressionConstraint ss ecr =  
minus (i\_subExpressionConstraint ss (first ecr))(i\_subExpressionConstraint ss (second ecr))

#### 4.1.8 subExpressionConstraint

`subExpressionConstraint` is interpreted as the interpretation of either a `simpleExpressionConstraint` or a `compoundExpressionConstraint`

`subExpressionConstraint = simpleExpressionConstraint / "(" ws (compoundExpressionConstraint / refinedExpressionConstraint) ws ")"`

$$\begin{aligned} \text{subExpressionConstraint} ::= & \\ & \text{subExpr\_simple} \langle \langle \text{simpleExpressionConstraint} \rangle \rangle \mid \\ & \text{subExpr\_compound} \langle \langle \text{compoundExpressionConstraint} \rangle \rangle \mid \\ & \text{subExpr\_refined} \langle \langle \text{refinedExpressionConstraint} \rangle \rangle \end{aligned}$$

*i\_subExpressionConstraint* :  
Substrate → subExpressionConstraint → Sctids\_or\_Error

∀ ss : Substrate; sec : subExpressionConstraint •  
i\_subExpressionConstraint ss sec =  
if sec ∈ ran subExpr\_simple  
  then i\_simpleExpressionConstraint ss (subExpr\_simple ~ sec)  
else if sec ∈ ran subExpr\_compound  
  then i\_compoundExpressionConstraint ss (subExpr\_compound ~ sec)  
else i\_refinedExpressionConstraint ss (subExpr\_refined ~ sec)

## 4.2 refinement

The interpretation of **refinement** is the interpretation of the **subRefinement**, **conjunctionGroup** or **disjunctionGroup**

$$\text{refinement} = \text{subRefinement} / \text{conjunctionGroup} / \text{disjunctionGroup}$$

*refinement* ::=  
     *refine\_subrefine*⟨⟨*subRefinement*⟩⟩ |  
     *refine\_conjg*⟨⟨*conjunctionGroup*⟩⟩ |  
     *refine\_disjg*⟨⟨*disjunctionGroup*⟩⟩

---

*i\_refinement* : *Substrate* → *refinement* → *Sctids\_or\_Error*

---

∀ *ss* : *Substrate*; *rfnment* : *refinement* • *i\_refinement* =  
**if** *rfnment* ∈ ran *refine\_subrefine*  
     **then** *i\_subRefinement* *ss* (*refine\_subrefine*~*rfnment*)  
**else if** *rfnment* ∈ ran *refine\_conjg*  
     **then** *i\_conjunctionGroup* *ss* (*refine\_conjg*~*rfnment*)  
**else** *i\_disjunctionGroup* *ss* (*refine\_disjg*~*rfnment*)

---

### 4.2.1 conjunctionGroup

$$\text{conjunctionGroup} = \text{subRefinement } 1^*(\text{conjunction subRefinement})$$

*conjunctionGroup* ==  
     *subRefinement* × seq<sub>1</sub>(*subRefinement*)

Apply the intersect operator to the interpretation of each *subRefinement*

---

*i\_conjunctionGroup* :  
     *Substrate* → *conjunctionGroup* → *Sctids\_or\_Error*

---

∀ *ss* : *Substrate*; *conjg* : *conjunctionGroup* •  
*i\_conjunctionGroup* *ss* *conjg* =  
     *applyToSequence* *ss* *i\_subRefinement* *intersect* *conjg*

---

### 4.2.2 disjunctionGroup

$$\text{disjunctionGroup} = \text{subRefinement } 1^*(\text{disjunction subRefinement})$$

$$\text{disjunctionGroup} == \\ \text{subRefinement} \times \text{seq}_1(\text{subRefinement})$$

Apply the union operator to the interpretation of each subRefinement

$i\_disjunctionGroup :$ $Substrate \rightarrow disjunctionGroup \rightarrow Sctids\_or\_Error$
$\forall ss : Substrate; disjg : disjunctionGroup \bullet$ $i\_disjunctionGroup ss disjg =$ $applyToSequence ss i\_subRefinement \text{ union } disjg$

### 4.2.3 subRefinement

The interpretation of a **subRefinement** is the interpretation of the corresponding **attributeSet**, **attributeGroup** or **refinement**.

subRefinement = attributeSet / attributeGroup / "(" ws refinement ws ")"?

$$\text{subRefinement} ::= \\ \text{subrefine\_attset} \langle\langle \text{attributeSet} \rangle\rangle \mid \\ \text{subrefine\_attgroup} \langle\langle \text{attributeGroup} \rangle\rangle \mid \\ \text{subrefine\_refinement} \langle\langle \text{refinement} \rangle\rangle$$

$i\_subRefinement :$ $Substrate \rightarrow subRefinement \rightarrow Sctids\_or\_Error$
$\forall ss : Substrate; subrefine : subRefinement \bullet$ $i\_subRefinement ss subrefine =$ $\text{if } subrefine \in \text{ran } subrefine\_attset$ $\quad \text{then } i\_attributeSet ss (subrefine\_attset \sim subrefine)$ $\text{else if } subrefine \in \text{ran } subrefine\_attgroup$ $\quad \text{then } i\_attributeGroup ss (subrefine\_attgroup \sim subrefine)$ $\text{else } i\_refinement ss (subrefine\_refinement \sim subrefine)$

### 4.3 attributeSet

attributeSet = subAttributeSet / conjunctionAttributeSet / disjunctionAttributeSet

$$\text{attributeSet} ::= \\ \text{attset\_subattset} \langle\langle \text{subAttributeSet} \rangle\rangle \mid \\ \text{attset\_conjattset} \langle\langle \text{conjunctionAttributeSet} \rangle\rangle \mid \\ \text{attset\_disjattset} \langle\langle \text{disjunctionAttributeSet} \rangle\rangle$$

$i\_attributeSet :$ $Substrate \rightarrow attributeSet \rightarrow Sctids\_or\_Error$
$\forall ss : Substrate; attset : attributeSet \bullet$ $i\_attributeSet ss attset =$ $\text{if } attset \in \text{ran } attset\_subattset$ $\quad \text{then } i\_subAttributeSet ss (attset\_subattset \sim attset)$ $\text{else if } attset \in \text{ran } attset\_conjattset$ $\quad \text{then } i\_conjunctionAttributeSet ss (attset\_conjattset \sim attset)$ $\text{else } i\_disjunctionAttributeSet ss (attset\_disjattset \sim attset)$

#### 4.3.1 conjunctionAttributeSet

conjunctionAttributeSet = subAttributeSet 1\*(conjunction subAttributeSet)

$conjunctionAttributeSet ==$   
 $subAttributeSet \times seq_1(subAttributeSet)$

Apply the intersect operator to the interpretation of each subAttributeSet

$i\_conjunctionAttributeSet :$ $Substrate \rightarrow conjunctionAttributeSet \rightarrow Sctids\_or\_Error$
$\forall ss : Substrate; conjaset : conjunctionAttributeSet \bullet$ $i\_conjunctionAttributeSet ss conjaset =$ $applyToSequence ss i\_subAttributeSet \text{ intersect } conjaset$

#### 4.3.2 disjunctionAttributeSet

disjunctionAttributeSet = subAttributeSet 1\*(disjunction subAttributeSet)

$disjunctionAttributeSet ==$   
 $subAttributeSet \times seq_1(subAttributeSet)$

Apply the union operator to the interpretation of each subAttributeSet

$i\_disjunctionAttributeSet :$ $Substrate \rightarrow disjunctionAttributeSet \rightarrow Sctids\_or\_Error$
$\forall ss : Substrate; disjaset : disjunctionAttributeSet \bullet$ $i\_disjunctionAttributeSet ss disjaset =$ $applyToSequence ss i\_subAttributeSet \text{ union } disjaset$

### 4.3.3 subAttributeSet

subAttributeSet = attribute / "(" ws attributeSet ws ")"

*subAttributeSet* ::=  
     *subaset\_attribute*⟨⟨*attribute*⟩⟩ |  
     *subaset\_attset*⟨⟨*attributeSet*⟩⟩

*i\_subAttributeSet* :  
     *Substrate* → *subAttributeSet* → *Sctids\_or\_Error*

---

∀ *ss* : *Substrate*; *subaset* : *subAttributeSet* •  
*i\_subAttributeSet ss subaset* =  
**if** *subaset* ∈ ran *subaset\_attribute*  
     **then** *i\_attribute ss* (*subaset\_attribute* ~ *subaset*)  
     **else** *i\_attributeSet ss* (*subaset\_attset* ~ *subaset*)

### 4.4 attributeGroup

attributeGroup = [cardinality ws] "{" ws attributeSet ws "}"

*attributeGroup* == *cardinality*[0 .. 1] × *attributeSet*

*i\_attributeGroup* : *Substrate* → *attributeGroup* → *Sctids\_or\_Error*

### 4.5 attribute

attribute = [cardinality ws] [reverseFlag ws] ws attributeName ws (concreteComparisonOperator ws concreteValue / expressionComparisonOperator ws expressionConstraintValue )  
     cardinality = "[" nonNegativeIntegerValue to (nonNegativeIntegerValue / many) "]"

*attribute* ::=  
     *attrib\_conc*⟨⟨*concreteAttribute*⟩⟩ |  
     *attrib\_expr*⟨⟨*expressionAttribute*⟩⟩

$i\_attribute :$ $Substrate \rightarrow attribute \rightarrow Sctids\_or\_Error$
$\forall ss : Substrate; att : attribute \bullet$ $i\_attribute\ ss\ att =$ <b>if</b> $att \in \text{ran } attrib\_conc$ <b>then</b> $i\_concreteAttribute\ ss\ (attrib\_conc \sim att)$ <b>else</b> $i\_expressionAttribute\ ss\ (attrib\_expr \sim att)$

For the sake of simplicity, we separate out the components of the concrete and expression constraints.

$unlimitedNat ::= num\langle\mathbb{N}\rangle \mid many$   
 $cardinality == \mathbb{N} \times unlimitedNat$   
 $[reverseFlag]$

#### 4.5.1 expressionAttribute

expressionComparisonOperator = "=" / "!=" / "<>"

expressionComparisonOperator ::= eco\_eq | eco\_neq

$expressionAttribute$ $card : cardinality[0..1]$ $reverse : reverseFlag[0..1]$ $name : attributeName$ $operator : expressionComparisonOperator$ $value : SCTID$
--

#### 4.5.2 concreteAttribute

concreteComparisonOperator = "=" / "!=" / "<>" / "<=" / "<" / ">=" / ">"  
concreteValue = QM stringValue QM / "#" numericValue  
stringValue = 1\*(anyNonEscapedChar / escapedChar)  
numericValue = decimalValue / integerValue

concreteComparisonOperator ::= cco\_eq | cco\_neq | cco\_leq | ccl\_lt | cco\_geq | cco\_gt  
concreteValue ::= stringValue | integerValue | decimalValue



<i>concreteAttribute</i> <i>card</i> : <i>cardinality</i> [0 .. 1] <i>name</i> : <i>attributeName</i> <i>operator</i> : <i>concreteComparisonOperator</i> <i>value</i> : <i>concreteValue</i>
---

The interpretation of a **concreteAttribute** selects the set of quads in the substrate that have an attribute in the set of attributes determined by the interpretation of **attributeName** having *CONCRETEVALUE* targets that meet the supplied comparison rules.

<i>i_concreteAttribute</i> : <i>Substrate</i> → <i>concreteAttribute</i> → <i>Quads_or_Error</i>
$\forall ss : \text{Substrate}; ca : \text{concreteAttribute} \bullet$ <i>i_concreteAttribute</i> <i>ss</i> <i>ca</i> = (let <i>attids</i> == <i>i_attributeName</i> <i>ss</i> <i>ca.name</i> • <i>i_concreteAttributeConstraint</i> <i>ss</i> <i>attids</i> <i>ca.operator</i> <i>ca.value</i> )

## 4.6 Group Cardinality

The interpretation of cardinality within a group impose additional constraints:

- [0 .. *n*] – the set of all substrate concept codes that have at least one group (entry) in the substrate relationships and, at most *n* matching entries in the same group
- [0 .. 0] – the set of all substrate concept codes that have at least one group (entry) in the substrate relationships and *no* matching entries
- [1..\*] – (default) at least one matching entry in the substrate relationships
- [*m*<sub>1</sub> .. *n*<sub>1</sub>]*op*[*m*<sub>2</sub> .. *n*<sub>2</sub>]*...* – set of substrate concept codes where there exists at least one group where all conditions are simultaneously true

The interpretation of a grouped cardinality is a function from a set of SC-TID's to the groups in which they were qualified.

The algorithm below partitions the input set of Quads by group and validates the cardinality on a per-group basis. Groups that pass are returned

**TODO:** This assumes that *q.t* is always type object. It doesn't say what to do if it is concrete **TODO:** the *quads\_to\_idgroups* function seems to express what is described below more simply

$i\_groupCardinality :$ $Quads\_or\_Error \rightarrow cardinality[0..1] \rightarrow IDGroups$
$\forall quads : Quads\_or\_Error; oc : cardinality[0..1]; uniqueGroups : \mathbb{P} GROUP;$ $quadsByGroup : GROUP \rightarrow \mathbb{P} Quad \mid$ $uniqueGroups = \{q : quads\_for quads \bullet q.g\} \wedge$ $quadsByGroup = \{g : uniqueGroups; q : \mathbb{P} Quad \mid$ $q = \{e : quads\_for quads \mid e.g = g\} \bullet g \mapsto (evalCardinality oc q)\} \bullet$ $i\_groupCardinality quads oc =$ $id\_groups \{sctid : SCTID; groups : \mathbb{P} GROUP \mid sctid \in \{q : \bigcup(\text{ran } quadsByGroup) \bullet$ $\text{if } quad\_direction quads = source\_direction \text{ then } q.s \text{ else } object \sim q.t\} \wedge$ $groups = \{g : \text{dom } quadsByGroup \mid (\exists q : quadsByGroup g \bullet$ $sctid = \text{if } quad\_direction quads = source\_direction \text{ then } q.s \text{ else } object \sim q.t)\} \bullet$ $sctid \mapsto groups\}$

## 4.7 Cardinality

**Interpretation:** *cardinality* is tested against a set of quads with the following rules:

1. Errors are propagated
2. No cardinality or passing cardinality returns the sources / targets of the set of quads
3. Otherwise return an empty set

$i\_cardinality :$ $cardinality[0..1] \rightarrow Quads\_or\_Error \rightarrow Sctids\_or\_Error$
$\forall card : cardinality[0..1]; quads : Quads\_or\_Error \bullet$ $i\_cardinality card quads =$ $\text{if } quads \in \text{ran } qerror$ $\text{ then } idgroups\_to\_sctids (quads\_to\_idgroups quads)$ $\text{ else }$ $idgroups\_to\_sctids (quads\_to\_idgroups (quad\_value (evalCardinality card (quads\_for quads), quad\_$

**evalCardinality** Evaluate the cardinality of an arbitrary set of type *T*.

- If the cardinality isn't supplied ( $\#opt\_cardinality = 0$ ), return the set.
  - If the number of elements is greater or equal to the minimum cardinality ( $first (head opt\_cardinality)$ ) then:
    - If the max cardinality is an integer ( $num \sim second (head opt\_cardinality)$ ) and it is greater than or equal to the number of elements or:
    - the max cardinality is not specified ( $second (head opt\_cardinality) = many$ )
- return the set
- Otherwise return  $\emptyset$

$[T]$
$evalCardinality : cardinality[0 \dots 1] \rightarrow \mathbb{P} T \rightarrow \mathbb{P} T$ $\forall opt\_cardinality : cardinality[0 \dots 1]; t : \mathbb{P} T \bullet$ $evalCardinality opt\_cardinality t =$ <b>if</b> $\#opt\_cardinality = 0 \vee$ $(\#t \geq first(head\ opt\_cardinality)) \wedge$ $(second(head\ opt\_cardinality) = many \vee$ $num^\sim(second(head\ opt\_cardinality)) \geq \#t)$ <b>then</b> $t$ <b>else</b> $\emptyset$

## 5 Substrate Interpretations

This section defines the interpretations that are realized against the substrate.

### 5.1 attributeName

**attributeName** is the interpretation of a **conceptReference** with the additional caveat that the SCTID(s) have to be substrate *attributeIds*

$attributeName = conceptReference$
------------------------------------

$attributeName == conceptReference$

$i\_attributeName :$ $Substrate \rightarrow attributeName \rightarrow Sctids\_or\_Error$
$\forall ss : Substrate; attName : attributeName \bullet$ $i\_attributeName ss attName =$ <b>(let</b> $attn == i\_conceptReference ss attName \bullet$ <b>if</b> $attn \in \text{ran } error$ <b>then</b> $attn$ <b>else if</b> $(result\_sctids\ attn) \subseteq ss.attributeIds$ <b>then</b> $attn$ <b>else</b> $error\ unknownAttributeId)$

### 5.2 attributeExpressionConstraint

**attributeExpressionConstraint** takes a substrate, an optional reverse flag, a set of attribute SCTIDs, an expression operator (equal or not equal) and a set of subject/target SCTIDS (depending on whether reverse flag is present) and returns a collection of quads that match / don't match the entry.

$ \begin{aligned} & i\_attributeExpressionConstraint : \\ & \quad Substrate \rightarrow reverseFlag[0..1] \rightarrow Sctids\_or\_Error \rightarrow \\ & \quad \quad expressionComparisonOperator \rightarrow Sctids\_or\_Error \rightarrow Quads\_or\_Error \end{aligned} $
$ \begin{aligned} & \forall ss : Substrate; rf : reverseFlag[0..1]; atts : Sctids\_or\_Error; \\ & \quad op : expressionComparisonOperator; subj\_or\_targets : Sctids\_or\_Error \bullet \\ & i\_attributeExpressionConstraint ss rf atts op subj\_or\_targets = \\ & \text{if } atts \in \text{ran error} \vee subj\_or\_targets \in \text{ran error} \\ & \quad \text{then } qfirstError\{atts, subj\_or\_targets\} \\ & \text{else if } \#args.rf = 0 \wedge op = eco\_eq \text{ then} \\ & \quad quad\_value(\{t : result\_sctids subj\_or\_targets; \\ & \quad \quad a : result\_sctids atts; rels : ss.relationships \mid \\ & \quad \quad object \sim rels.t = t \wedge rels.a = a \bullet rels\}, source\_direction) \\ & \text{else if } \#args.rf = 1 \wedge op = eco\_eq \text{ then} \\ & \quad quad\_value(\{s : result\_sctids subj\_or\_targets; \\ & \quad \quad a : result\_sctids atts; rels : ss.relationships \mid \\ & \quad \quad rels.s = s \wedge rels.t \in \text{ran object} \wedge rels.a = a \bullet rels\}, targets\_direction) \\ & \text{else if } \#args.rf = 0 \wedge op = eco\_neq \text{ then} \\ & \quad quad\_value(\{t : result\_sctids subj\_or\_targets; \\ & \quad \quad a : result\_sctids atts; rels : ss.relationships \mid \\ & \quad \quad object \sim rels.t \neq t \wedge rels.a = a \bullet rels\}, source\_direction) \\ & \text{else} \\ & \quad quad\_value(\{s : result\_sctids subj\_or\_targets; \\ & \quad \quad a : result\_sctids atts; rels : ss.relationships \mid \\ & \quad \quad rels.s \neq s \wedge rels.t \in \text{ran object} \wedge rels.a = a \bullet rels\}, targets\_direction) \end{aligned} $

### 5.3 concreteAttributeConstraint

$ \begin{aligned} & i\_concreteAttributeConstraint : \\ & \quad Substrate \rightarrow Sctids\_or\_Error \rightarrow concreteComparisonOperator \rightarrow \\ & \quad \quad concreteValue \rightarrow Quads\_or\_Error \end{aligned} $
$ \begin{aligned} & \forall ss : Substrate; atts : Sctids\_or\_Error; op : concreteComparisonOperator; \\ & \quad val : concreteValue \bullet \\ & i\_concreteAttributeConstraint ss atts op val = \\ & \text{if } atts \in \text{ran error} \\ & \quad \text{then } qerror(error \sim atts) \\ & \text{else } quad\_value\{rels : ss.relationships \mid rels.a \in (result\_sctids atts) \wedge rels.t \in \text{ran concrete} \wedge val \in \text{concrete}\} \end{aligned} $

  

$ \begin{aligned} & concreteMatch : \\ & \quad \mathbb{P} CONCRETEVALUE \rightarrow concreteComparisonOperator \rightarrow \mathbb{P} concreteValue \end{aligned} $
---

**Interpretation:** Apply the substrate descendants (*descs*) or ancestors (*ancs*)

function to a set of SCTID's in the supplied *Sctids\_or\_Error*. Error conditions are propagated.

$ \begin{array}{l} i\_constraintOperator : \\ \quad Substrate \rightarrow constraintOperator[0..1] \rightarrow Sctids\_or\_Error \rightarrow Sctids\_or\_Error \\ completeFun : (SCTID \rightarrow \mathbb{P} SCTID) \rightarrow SCTID \rightarrow \mathbb{P} SCTID \\ \hline \forall ss : Substrate; oco : constraintOperator[0..1]; subresult : Sctids\_or\_Error \bullet \\ i\_constraintOperator ss oco subresult = \\ \quad \text{if } error \sim subresult \in ERROR \vee \#oco = 0 \\ \quad \quad \text{then } subresult \\ \quad \text{else if } head\ oco = descendantOrSelfOf \\ \quad \quad \text{then } ok(\bigcup \{id : result\_sctids\ subresult \bullet \\ \quad \quad \quad completeFun ss.descendants id\} \cup result\_sctids\ subresult) \\ \quad \text{else if } head\ oco = descendantOf \\ \quad \quad \text{then } ok(\bigcup \{id : result\_sctids\ subresult \bullet \\ \quad \quad \quad completeFun ss.descendants id\}) \\ \quad \text{else if } head\ oco = ancestorOrSelfOf \\ \quad \quad \text{then } ok(\bigcup \{id : result\_sctids\ subresult \bullet \\ \quad \quad \quad completeFun ss.ancestors id\} \cup result\_sctids\ subresult) \\ \quad \text{else } ok(\bigcup \{id : result\_sctids\ subresult \\ \quad \quad \bullet completeFun ss.ancestors id\}) \\ \hline \forall f : (SCTID \rightarrow \mathbb{P} SCTID); id : SCTID \bullet completeFun f id = \\ \quad \text{if } id \in \text{dom } f \text{ then } f\ id \text{ else } \emptyset \end{array} $
--

## 5.4 FocusConcept

`focusConcept` = `[memberOf]` `conceptReference`

### 5.4.1 focusConcept

`focusConcept` is either a simple concept reference or the interpretation of the `memberOf` function applied to a concept reference.

$$\begin{array}{l}
focusConcept ::= \\
\quad focusConcept\_m \langle\langle conceptReference \rangle\rangle \mid \\
\quad focusConcept\_c \langle\langle conceptReference \rangle\rangle
\end{array}$$

**Interpretation:** If `memberOf` is present the interpretation of `focusConcept` is union the interpretation of `memberOf` applied to each element in the interpretation of `conceptReference`. If `memberOf` isn't part of the spec, the interpretation is the interpretation of `conceptReference` itself

$i\_focusConcept : Substrate \rightarrow focusConcept \rightarrow Sctids\_or\_Error$
$\forall ss : Substrate; fc : focusConcept \bullet$ $i\_focusConcept ss fc =$ <b>if</b> $focusConcept\_c \sim fc \in conceptReference$ <b>then</b> $i\_conceptReference ss (focusConcept\_c \sim fc)$ <b>else</b> $i\_memberOf ss (i\_conceptReference ss (focusConcept\_m \sim fc))$

#### 5.4.2 memberOf

**memberOf** returns the union of the application of the substrate *refset* function to each of the supplied reference set identifiers. An error is returned if (a) *refsetids* already has an error or (b) one or more of the refset identifiers aren't substrate *refsetIds*

$i\_memberOf : Substrate \rightarrow Sctids\_or\_Error \rightarrow Sctids\_or\_Error$ $i\_refset : Substrate \rightarrow SCTID \rightarrow Sctids\_or\_Error$
$\forall ss : Substrate; refsetids : Sctids\_or\_Error \bullet$ $i\_memberOf ss refsetids =$ <b>if</b> $refsetids \in \text{ran error}$ <b>then</b> $refsetids$ <b>else</b> $bigunion\{sctid : result\_sctids refsetids \bullet i\_refset ss sctid\}$
$\forall ss : Substrate; sctid : SCTID \bullet$ $i\_refset ss sctid =$ <b>if</b> $sctid \notin ss.refsetIds$ <b>then</b> $error unknownRefsetId$ <b>else if</b> $sctid \in \text{dom } ss.refsetIds$ <b>then</b> $ok (ss.refsets sctid)$ <b>else</b> $ok \emptyset$

### 5.5 ConceptReferences

#### 5.5.1 conceptId

**conceptId** = **sctId**

**Interpretation:** **conceptId** is interpreted as *SCTID* that it represents. For our purposes, all **conceptIds** are considered valid, so this is a bijection.

[*conceptId*]

$i\_conceptId : conceptId \rightarrow SCTID$
--

### 5.5.2 conceptReference

```
conceptReference = conceptId [ "|" Term "|"]
conceptId = sctId
```

**Interpretation:** `conceptReference` is interpreted as the *set of SCTIDs* that are equivalent to the supplied SCTID if it is known concepts, *c*, in the substrate. If it isn't in the list of known concepts otherwise as the *unknownConceptReference* error. The `Term` part of `conceptReference` is ignored.

*conceptReference == conceptId*

$i\_conceptReference : Substrate \rightarrow conceptReference \rightarrow Sctids\_or\_Error$
$\forall ss : Substrate; c : conceptReference \bullet i\_conceptReference ss c =$ $(\text{let } sctid == i\_conceptId c \bullet$ $\text{if } sctid \in ss.concepts \text{ then } ok(ss.equivalent\_concepts sctid)$ $\text{else error } unknownConceptReference)$

## 6 Glue and Helper Functions

This section carries various type transformations and error checking functions

### 6.1 Types

- **direction** – an indicator whether a collection of quads was determined as source to target (*source\_direction*) or target to source (*targets\_direction*)
- **Quads\_or\_Error** – a collection of *Quads* or an error condition. If it is a collection *Quads*, it also carries a direction indicator that determines whether it represents a set of sources or targets.
- **IDGroups** – a map from *SCTIDs* to the *GROUP* they were in when they passed if successful, otherwise an error indication.

*direction* ::= *source\_direction* | *targets\_direction*  
*Quads\_or\_Error* ::= *quad\_value*⟨⟨ $\mathbb{P} Quad \times direction$ ⟩⟩ | *qerror*⟨⟨*ERROR*⟩⟩  
*IDGroups* ::= *id\_groups*⟨⟨*SCTID*  $\rightarrow$   $\mathbb{P} GROUP$ ⟩⟩ | *gerror*⟨⟨*ERROR*⟩⟩

### 6.2 Result transformations

- **result\_sctids** – the set of *SCTIDs* in *Sctids\_or\_Error* or the empty set if there is an error
- **quads\_for** – the set of quads in a *Quads\_or\_Error* or an empty set if there is an error
- **quad\_direction** – the direction of a *Quads\_or\_Error* result. Undefined if error

- **to\_idGroups** – the *SCTID* to *GROUP* part of in an id group or an empty map if there is error
- **quads\_to\_idgroups** – convert a set of quads into a set of id groups using the following rules:
  - If the set of quads has an error, propagate it
  - If the quad direction is *source\_direction* (target to source) a list of unique relationship subjects and, for each subjects, the set of different groups it appears as a subject in
  - Otherwise return a list of relationship target sctids and, for each target, the set of different groups it appears as a target in.
- **idgroups\_to\_sctids** – remove the groups and return an *Sctids\_or\_Error* for the ids

---

```

result_sctids : Sctids_or_Error →  $\mathbb{P}$  SCTID
quads_for : Quads_or_Error →  $\mathbb{P}$  Quad
quad_direction : Quads_or_Error → direction
to_idGroups : IDGroups → SCTID →  $\mathbb{P}$  GROUP
quads_to_idgroups : Quads_or_Error → IDGroups
idgroups_to_sctids : IDGroups → Sctids_or_Error

```

---

```

∀ r : Sctids_or_Error • result_sctids r =
  if error~r ∈ ERROR then ∅
  else ok~r

∀ q : Quads_or_Error • quads_for q =
  if qerror~q ∈ ERROR then ∅
  else first (quad_value~q)

∀ q : Quads_or_Error • quad_direction q =
  second (quad_value~q)

∀ g : IDGroups • to_idGroups g =
  if gerror~g ∈ ERROR then ∅
  else id_groups~g

∀ q : Quads_or_Error • quads_to_idgroups q =
  if qerror~q ∈ ERROR then gerror(qerror~q)
  else if quad_direction q = source_direction
    then id_groups {s : SCTID | (∃ qr : quads_for q • s = qr.s) •
      s ↦ {qr : quads_for q • qr.g}}
    else
      id_groups {t : SCTID | (∃ qr : quads_for q • t = object~qr.t) •
        t ↦ {qr : quads_for q • qr.g}}

∀ g : IDGroups • idgroups_to_sctids g =
  if g ∈ ran gerror then ok ∅
  else ok (dom(id_groups~g))

```

---



Definition of the various functions that are performed on the result type.

- **firstError** – aggregate one or more *Sctids\_or\_Error* types, at least one of which carries an error and merge them into a single *Sctid\_or\_Error* instance propagating at least one of the errors (Not fully defined)
- **qfirstError** – convert two *Sctids\_or\_Error* types, into a *Quads\_or\_Error* propagating at least one of the errors. (not fully defined)
- **union** – return the union of two *Sctids\_or\_Error* types, propagating errors if they exist, else returning the union of the SCTID sets.
- **intersect** – return the intersection of two *Sctids\_or\_Error* types, propagating errors if they exist, else returning the intersection of the SCTID sets.
- **minus** – return the difference of one *Sctids\_or\_Error* type and a second, propagating errors if they exist, else returning the set of SCTID's in the first set that aren't in the second.
- **bigunion** – return the union of a set of *Sctids\_or\_Error* types, propagating errors if they exist, else returning the union of all of the SCTID sets.
- **bigintersect** – return the intersection a set of *Sctids\_or\_Error* types, propagating errors if they exist, else returning the intersection of all of the SCTID sets.

$firstError : \mathbb{P} Sctids\_or\_Error \rightarrow Sctids\_or\_Error$ $qfirstError : \mathbb{P} Sctids\_or\_Error \rightarrow Quads\_or\_Error$ $union, intersect, minus : Sctids\_or\_Error \rightarrow Sctids\_or\_Error \rightarrow Sctids\_or\_Error$ $bigunion, bigintersect : \mathbb{P} Sctids\_or\_Error \rightarrow Sctids\_or\_Error$
$\forall x, y : Sctids\_or\_Error \bullet union\ x\ y =$ <b>if</b> $x \in \text{ran } error \vee y \in \text{ran } error$ <b>then</b> $firstError\ \{x, y\}$ <b>else</b> $ok\ ((ok \sim x) \cup (ok \sim y))$ $\forall x, y : Sctids\_or\_Error \bullet intersect\ x\ y =$ <b>if</b> $x \in \text{ran } error \vee y \in \text{ran } error$ <b>then</b> $firstError\ \{x, y\}$ <b>else</b> $ok\ ((ok \sim x) \cap (ok \sim y))$ $\forall x, y : Sctids\_or\_Error \bullet minus\ x\ y =$ <b>if</b> $x \in \text{ran } error \vee y \in \text{ran } error$ <b>then</b> $firstError\ \{x, y\}$ <b>else</b> $ok\ ((ok \sim x) \setminus (ok \sim y))$ $\forall rs : \mathbb{P} Sctids\_or\_Error \bullet bigunion\ rs =$ <b>if</b> $\exists r : rs \bullet r \in \text{ran } error$ <b>then</b> $firstError\ rs$ <b>else</b> $ok\ (\bigcup \{r : rs \bullet result\_sctids\ r\})$ $\forall rs : \mathbb{P} Sctids\_or\_Error \bullet bigintersect\ rs =$ <b>if</b> $\exists r : rs \bullet r \in \text{ran } error$ <b>then</b> $firstError\ rs$ <b>else</b> $ok\ (\bigcap \{r : rs \bullet result\_sctids\ r\})$

## 7 Appendix 1

Representing optional elements of type  $T$ . Representing it as a sequence allows us to determine absence by  $\#T = 0$  and the value by  $head\ T$ .

$$T[0..1] == \{s : seq\ T \mid \#s \leq 1\}$$

## 8 Appendix 2

A generic function that takes:

- A substrate
- A function that takes a substrate, a sequence of type  $T$  and returns  $Sctids\_or\_Error$  (example:  $i\_subExpressionConstraint$ )
- An operator that takes two  $Sctids\_or\_Error$  and returns a combination (example:  $union$ )
- A structure of the form  $\text{"} T \times seq_1\ T$

And returns  $Sctids\_or\_Error$

In the formalization below, *first seq\_e* refers to the left hand side of the  $T \times \text{seq}_1 T$  and *second seq\_e* to the right hand side. *head(second seq\_e)* refers to the first element in the sequence and *tail(second seq\_e)* refers to the remaining elements in the sequence, which may be empty ( $\langle \rangle$ ).

$$\begin{array}{l}
\text{[T]} \\
\text{applyToSequence} : \text{Substrate} \rightarrow (\text{Substrate} \rightarrow T \rightarrow \text{Sctids\_or\_Error}) \rightarrow \\
\quad (\text{Sctids\_or\_Error} \rightarrow \text{Sctids\_or\_Error} \rightarrow \text{Sctids\_or\_Error}) \rightarrow \\
\quad (T \times \text{seq}_1 T) \rightarrow \text{Sctids\_or\_Error} \\
\forall ss : \text{Substrate}; f : (\text{Substrate} \rightarrow T \rightarrow \text{Sctids\_or\_Error}); \\
\quad op : (\text{Sctids\_or\_Error} \rightarrow \text{Sctids\_or\_Error} \rightarrow \text{Sctids\_or\_Error}); \\
\quad seq\_e : (T \times \text{seq}_1 T) \bullet \\
\text{applyToSequence } ss \text{ } f \text{ } op \text{ } seq\_e = \\
\text{if } \text{tail}(\text{second } seq\_e) = \langle \rangle \text{ then} \\
\quad op(f \text{ } ss \text{ } (\text{first } seq\_e))(f \text{ } ss \text{ } (\text{head}(\text{second } seq\_e))) \\
\text{else} \\
\quad op(f \text{ } ss \text{ } (\text{first } seq\_e))(f \text{ } ss \text{ } ((\text{head}(\text{second } seq\_e))(\text{tail}(\text{second } seq\_e))))
\end{array}$$