A Declarative Semantics for SNOMED CT Expression Constraints

March 15, 2015

Contents

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
- **decimalValue** – a decimal number
- **stringValue** – a string literal
- **groupId** – a role group identifier
- $\mathbb{N}$ – a non-negative integer (built in to Z)
- $\mathbb{Z}$ – an integer (built in to Z)

$$[sctId, term, decimalValue, stringValue, groupId]$$

We will also need to recognize some well known identifiers: the *is_a* attribute, the *zero_group* and *attribute_concept*, and *refset_concept* the parents of all attributes and all refsets respectively.

$$
\begin{array}{|l}
is\_a : sctId \\
zero\_group : groupId \\
attribute\_concept : sctId \\
refset\_concept : sctId
\end{array}
$$

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 decimal literal
- **target** – the target of a relationship that is either an *sctId* or a *concreteValue*

$$
\begin{array}{l}
concreteValue ::= \\
\quad cv\_string\langle\langle stringValue\rangle\rangle \mid cv\_integer\langle\langle \mathbb{Z}\rangle\rangle \mid cv\_decimal\langle\langle decimalValue\rangle\rangle \\
target ::= t\_sctid\langle\langle sctId\rangle\rangle \mid t\_concrete\langle\langle concreteValue\rangle\rangle
\end{array}
$$

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

$$
\begin{array}{|l}
s : sctId \\
a : sctId \\
t : target \\
g : groupId \\
\hline
a = is\_a \Rightarrow (g = zero\_group \land t \in \operatorname{ran} t\_sctid)
\end{array}
$$

2.1.1 conceptReference

The root of the expression constraint language is concept references – textual representations SNOMED CT identifiers accompanied by an optional *term* that conveys their intended meaning to the human reader. `term` is ignored for the purposes of interpretation.

```
conceptReference = conceptId [ws "—" ws term ws "—"]
conceptId = sctId
```

$conceptId == sctId$
$conceptReference == conceptId \times term[0\,..\,1]$
$attributeName == conceptReference$

2.2 Substrate

A substrate consists of:

- **concepts** The set of *sctId*s (concepts) that are considered valid in the context of the substrate.
- **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

A substrate also implements three functions:

- **i_conceptReference** The interpretation of a concept reference. This function can return a (possibly empty) set of sctId's or an error.
- **i_attributeName** The interpretation of an attribute name. This function can return a (possibly empty) set of sctId's or an error.
- **i_refsetId** The interpretation of a refset identifier. This function can return a (possibly empty) set of sctId's or an error.

The formal definition of substrate follows. The expressions below assert that:

1. All *relationship* sources, attributes and sctId targets are in *concepts*.
2. There is a *parentsOf* entry for every substrate *concept*.
3. Every sctId that can be returned by the *parentsOf* function is a *concept*.
4. Every *is_a* relationship entry is represented in the *parentsOf* function. Note that there can be additional entries represented in the *parentsOf* function that aren't in the relationships table.
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. All equivalent concepts are in *concepts*.
8. If two concepts (c2 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

9. Every refset is a substrate *concept*
10. Every member of a refset is a substrate *concept*
11. *childrenOf* is the inverse of *parentsOf*, where any concept that doesn't appear a parent has no (the emptyset) children.
12. *descendants* is the transitive closure of the *childrenOf* function
13. *ancestors* is the transitive closure of the *parentsOf* function
14. No concept can be its own ancestor (or, by inference, descendant)
15. The *i_conceptReference*, *i_attributeName* and *i_refsetId* functions are defined for all possible *conceptReferences* and *attributeNames* (because they are complete functions).
16. All $sctId's$ that are produced by the The *i_conceptReference*, *i_attributeName* and *i_refsetId* functions are substrate *concepts*.

Substrate

$concepts : \mathbb{P}\ sctId$
$relationships : \mathbb{P}\ Quad$
$parentsOf : sctId \pfun \mathbb{P}\ sctId$
$equivalent\_concepts : sctId \pfun \mathbb{P}\ sctId$
$refsets : sctId \pfun \mathbb{P}\ sctId$

$childrenOf : sctId \pfun \mathbb{P}\ sctId$
$descendants : sctId \pfun \mathbb{P}\ sctId$
$ancestors : sctId \pfun \mathbb{P}\ sctId$

$i\_conceptReference : conceptReference \to Sctids\_or\_Error$
$i\_attributeName : attributeName \to Sctids\_or\_Error$
$i\_refsetId : sctId \to Sctids\_or\_Error$

$\forall\ rel : relationships \bullet rel.s \in concepts \land rel.a \in concepts \land$
$\quad (rel.t \in \operatorname{ran} t\_sctid \Rightarrow t\_sctid^{\sim} rel.t \in concepts)$

$\operatorname{dom} parentsOf = concepts$
$\bigcup(\operatorname{ran} parentsOf) \subseteq concepts$

$\forall\ r : relationships \bullet r.a = is\_a \Rightarrow (t\_sctid^{\sim} r.t) \in parentsOf\ r.s$

$\operatorname{dom} equivalent\_concepts = concepts$
$\bigcup(\operatorname{ran} equivalent\_concepts) \subseteq concepts$
$\forall\ c : concepts \bullet c \in equivalent\_concepts\ c$
$\forall\ c1, c2 : concepts \mid c2 \in (equivalent\_concepts\ c1) \bullet$
$\quad parentsOf\ c1 = parentsOf\ c2 \land$
$\quad \{r : relationships \mid r.s = c1\} = \{r : relationships \mid r.s = c2\} \land$
$\quad \{r : relationships \mid r.a = c1\} = \{r : relationships \mid r.a = c2\} \land$
$\quad \{r : relationships \mid t\_sctid^{\sim} r.t = c1\} = \{r : relationships \mid t\_sctid^{\sim} r.t = c2\} \land$
$\quad c1 \in \operatorname{dom} refsets \Leftrightarrow c2 \in \operatorname{dom} refsets \land$
$\quad c1 \in \operatorname{dom} refsets \Rightarrow refsets\ c1 = refsets\ c2 \land$
$\quad (\forall\ rsd : \operatorname{ran} refsets \bullet c1 \in rsd \Leftrightarrow c2 \in rsd)$

$\operatorname{dom} refsets \subseteq concepts$
$\bigcup(\operatorname{ran} refsets) \subseteq concepts$

$\operatorname{dom} childrenOf = concepts$
$\forall\ s, t : concepts \bullet t \in parentsOf\ s \Leftrightarrow s \in childrenOf\ t$
$\forall\ c : concepts \mid c \notin \bigcup(\operatorname{ran} childrenOf) \bullet childrenOf\ c = \emptyset$

$\forall\ s : concepts \bullet$
$\quad descendants\ s = childrenOf\ s \cup \bigcup\{t : childrenOf\ s \bullet descendants\ t\}$
$\forall\ t : concepts \bullet$
$\quad ancestors\ t = parentsOf\ t \cup \bigcup\{s : parentsOf\ t \bullet ancestors\ s\}$
$\forall\ t : concepts \bullet t \notin ancestors\ t$

$\forall\ cr\_interp : \operatorname{ran} i\_conceptReference \mid cr\_interp \in \operatorname{ran} ok \bullet$
$\quad result\_sctids\ cr\_interp \subseteq concepts$
$\forall\ att\_interp : \operatorname{ran} i\_attributeName \mid att\_interp \in \operatorname{ran} ok \bullet$
$\quad result\_sctids\ att\_interp \subseteq concepts$
$\forall\ refset\_interp : \operatorname{ran} i\_refsetId \mid refset\_interp \in \operatorname{ran} ok \bullet$
$\quad result\_sctids\ refset\_interp \subseteq concepts$

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:

- If a conceptReference is not in the substrate concepts it returns an error, otherwise the set of equivalent concepts
- If an attribute name is not in the substrate concepts or is not a descendant of the attribute_concept it returns an error, otherwise the set of equivalent attributes
- If a concept reference that is a target of a memberOf function is not in the substrate concepts or is not a descendant of the refset_concept it returns an error, otherwise the set of equivalent refset identifiers

strict_substrate

Substrate

$\forall\, cr : conceptReference \bullet i\_conceptReference\ cr =$
if $first\ cr \notin concepts$ **then** $error\ unknownConceptReference$
else $ok\ (equivalent\_concepts\ (first\ cr))$

$\forall\, an : attributeName \bullet i\_attributeName\ an =$
(**let** $rslt == i\_conceptReference\ an \bullet$
if $rslt \in \mathrm{ran}\ error$ **then** $rslt$
else if $result\_sctids\ rslt \subseteq (descendants\ attribute\_concept)$
then $rslt$
else $error\ unknownAttributeId$)

$\forall\, rsid : sctId \bullet i\_refsetId\ rsid =$
if $rsid \in descendants\ refset\_concept$
then $ok\ \{rsid\}$
else $error\ unknownRefsetId$

2.2.3 Permissive Substrate

(fill in options)

3 Interpretation of Expression Constraints

An `expressionConstraint` is interpreted in the context of a *Substrate* and returns a set of *sctId*s or an error indicator.

3.1 Interpretation Output

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*

$$ERROR ::= unknownConceptReference \mid unknownAttributeId \mid unknownRefsetId$$

$$Sctids\_or\_Error ::= ok\langle\langle \mathbb{P}\, sctId \rangle\rangle \mid error\langle\langle ERROR \rangle\rangle$$

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.

3.2 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.

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

$$\begin{array}{|l} i\_expressionConstraint : \\ \quad Substrate \rightarrow expressionConstraint \rightarrow Sctids\_or\_Error \\ \hline \forall\, ss : Substrate;\ ec : expressionConstraint \bullet i\_expressionConstraint\ ss\ ec = \\ \textbf{if}\ ec \in \mathrm{ran}\ expcons\_refined \\ \quad \textbf{then}\ i\_refinedExpressionConstraint\ ss\ (expcons\_refined^{\sim}\ ec) \\ \textbf{else}\ i\_unrefinedExpressionConstraint\ ss\ (expcons\_unrefined^{\sim}\ ec) \end{array}$$

3.2.1 unrefinedExpressionConstraint

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

unrefinedExpressionConstraint = compoundExpressionConstraint / simpleExpressionConstraint

$$unrefinedExpressionConstraint ::= unrefined\_compound\langle\langle compoundExpressionConstraint\rangle\rangle \mid unrefined\_simple\langle\langle simpleExpressionConstraint\rangle\rangle$$

$$i\_unrefinedExpressionConstraint : Substrate \to unrefinedExpressionConstraint \to Sctids\_or\_Error$$

$$\forall\, ss : Substrate;\ uec : unrefinedExpressionConstraint \bullet$$
$$i\_unrefinedExpressionConstraint\ ss\ uec =$$
$$\textbf{if}\ uec \in \operatorname{ran} unrefined\_compound$$
$$\quad \textbf{then}\ i\_compoundExpressionConstraint\ ss\ (unrefined\_compound^{\sim} uec)$$
$$\textbf{else}\ i\_simpleExpressionConstraint\ ss\ (unrefined\_simple^{\sim} uec)$$

3.2.2 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.

$$refinedExpressionConstraint == unrefinedExpressionConstraint \times refinement$$
$$[refinedExpressionConstraint']$$

$$i\_refinedExpressionConstraint : Substrate \to refinedExpressionConstraint \to Sctids\_or\_Error$$

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

$$i\_refinedExpressionConstraint' : Substrate \to refinedExpressionConstraint' \to Sctids\_or\_Error$$

3.2.3 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 / ancestorOrSelfOf / ancestorOf

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

$$i\_simpleExpressionConstraint : Substrate \rightarrow simpleExpressionConstraint \rightarrow Sctids\_or\_Error$$

$$\forall\, ss : Substrate;\ sec : simpleExpressionConstraint \bullet$$
$$i\_simpleExpressionConstraint\ ss\ sec = i\_constraintOperator\ ss\ (first\ sec)\ (i\_focusConcept\ ss\ (second\ sec))$$

3.2.4 compoundExpressionConstraint

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

> compoundExpressionConstraint = F1mU7 ExpressionConstraint / disjunctionExpressionConstraint / exclusionExpressionConstraint / "(" ws compoundExpressionConstraint ws ")"

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

$$i\_compoundExpressionConstraint : Substrate \rightarrow compoundExpressionConstraint \rightarrow Sctids\_or\_Error$$

$$\forall\, ss : Substrate;\ cec : compoundExpressionConstraint \bullet$$
$$i\_compoundExpressionConstraint\ ss\ cec =$$
$$\textbf{if}\ cec \in \operatorname{ran} compound\_conj\ \textbf{then}\ i\_conjunctionExpressionConstraint\ ss\ (compound\_conj^{\sim}\ cec)$$
$$\textbf{else if}\ cec \in \operatorname{ran} compound\_disj\ \textbf{then}\ i\_disjunctionExpressionConstraint\ ss\ (compound\_disj^{\sim}\ cec)$$
$$\textbf{else}\ i\_exclusionExpressionConstraint\ ss\ (compound\_excl^{\sim}\ cec)$$

The signature below is used because the definition of `compountExpressionConstraint` is recursive

$$i\_compoundExpressionConstraint' : Substrate \rightarrow compoundExpressionConstraint \rightarrow Sctids\_or\_Error$$

3.2.5 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)
```

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

Apply the intersection operator to the interpretation of each subExpressionConstraint

$$i\_conjunctionExpressionConstraint : Substrate \rightarrow conjunctionExpressionConstraint \rightarrow Sctids\_or\_Error$$

$$\forall ss : Substrate;\ cecr : conjunctionExpressionConstraint \bullet$$
$$i\_conjunctionExpressionConstraint\ ss\ cecr = applyToSequence\ ss\ i\_subExpressionConstraint\ intersect\ cecr$$

3.2.6 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)
```

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

Apply the union operator to the interpretation of each subExpressionConstraint

$$
\begin{array}{|l}
\hline
i\_disjunctionExpressionConstraint : \\
\quad Substrate \rightarrow disjunctionExpressionConstraint \rightarrow Sctids\_or\_Error \\
\hline
\forall\, ss : Substrate;\ decr : disjunctionExpressionConstraint \bullet \\
i\_disjunctionExpressionConstraint\ ss\ decr = \\
\quad applyToSequence\ ss\ i\_subExpressionConstraint\ union\ decr \\
\hline
\end{array}
$$

3.2.7 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 subExpression-
Constraint
```

$$
exclusionExpressionConstraint == subExpressionConstraint \times subExpressionConstraint
$$

$$
\begin{array}{|l}
\hline
i\_exclusionExpressionConstraint : \\
\quad Substrate \rightarrow exclusionExpressionConstraint \rightarrow Sctids\_or\_Error \\
\hline
\forall\, ss : Substrate;\ ecr : exclusionExpressionConstraint \bullet \\
i\_exclusionExpressionConstraint\ ss\ ecr = \\
\quad minus\ (i\_subExpressionConstraint\ ss\ (first\ ecr))(i\_subExpressionConstraint\ ss\ (second\ ecr)) \\
\hline
\end{array}
$$

3.2.8 subExpressionConstraint

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

```
subExpressionConstraint = simpleExpressionConstraint / "(" ws (compoundExpressionCon-
straint / refinedExpressionConstraint) ws ")"
```

$$
\begin{array}{l}
subExpressionConstraint ::= \\
\quad subExpr\_simple\langle\langle simpleExpressionConstraint\rangle\rangle \mid \\
\quad subExpr\_compound\langle\langle compoundExpressionConstraint\rangle\rangle \mid \\
\quad subExpr\_refined\langle\langle refinedExpressionConstraint'\rangle\rangle
\end{array}
$$

$$
\begin{array}{l}
i\_subExpressionConstraint : \\
\qquad Substrate \rightarrow subExpressionConstraint \rightarrow Sctids\_or\_Error \\
\hline
\forall\, ss : Substrate;\ sec : subExpressionConstraint \bullet \\
i\_subExpressionConstraint\ ss\ sec = \\
\textbf{if}\ sec \in \operatorname{ran} subExpr\_simple \\
\quad \textbf{then}\ i\_simpleExpressionConstraint\ ss\ (subExpr\_simple^{\sim} sec) \\
\textbf{else if}\ sec \in \operatorname{ran} subExpr\_compound \\
\quad \textbf{then}\ i\_compoundExpressionConstraint'\ ss\ (subExpr\_compound^{\sim} sec) \\
\textbf{else}\ i\_refinedExpressionConstraint'\ ss\ (subExpr\_refined^{\sim} sec)
\end{array}
$$

3.3 refinement

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

```
refinement = subRefinement [conjunctionRefinementSet / disjunctionRefinementSet]
```

$$
refinement == subRefinement \times refinementConjunctionOrDisjunction[0\,..\,1]
$$
$$
[refinement']
$$

$$
\begin{array}{l}
refinementConjunctionOrDisjunction ::= \\
\quad refine\_conjset\langle\langle conjunctionRefinementSet\rangle\rangle \mid \\
\quad refine\_disjset\langle\langle disjunctionRefinementSet\rangle\rangle
\end{array}
$$

$$
\begin{array}{l}
i\_refinement : Substrate \rightarrow refinement \rightarrow Sctids\_or\_Error \\
\hline
\forall\, ss : Substrate;\ rfnment : refinement \bullet \\
i\_refinement\ ss\ rfnment = \\
(\textbf{let}\ lhs == i\_subRefinement\ ss\ (first\ rfnment);\ rhs == second\ rfnment \bullet \\
\textbf{if}\ \#rhs = 0 \\
\quad \textbf{then}\ lhs \\
\textbf{else if}\ (head\ rhs) \in \operatorname{ran} refine\_conjset \\
\quad \textbf{then}\ intersect\ lhs\ (i\_conjunctionRefinementSet\ ss\ (refine\_conjset^{\sim}(head\ rhs))) \\
\textbf{else}\ union\ lhs\ (i\_disjunctionRefinementSet\ ss\ (refine\_disjset^{\sim}(head\ rhs))))
\end{array}
$$

$$
\begin{array}{l}
i\_refinement' : Substrate \rightarrow refinement' \rightarrow Sctids\_or\_Error
\end{array}
$$

3.3.1 conjunctionRefinementSet

```
conjunctionRefinementSet = 1*(ws conjunction ws subRefinement)
```

$conjunctionRefinementSet == \text{seq}_1\ subRefinement$

Apply the intersect operator to the interpretation of each subRefinement

$$
\begin{array}{l}
i\_conjunctionRefinementSet : \\
\quad Substrate \rightarrow conjunctionRefinementSet \rightarrow Sctids\_or\_Error \\
\hline
\forall\ ss : Substrate;\ conjset : conjunctionRefinementSet \bullet \\
i\_conjunctionRefinementSet\ ss\ conjset = \\
\textbf{if}\ tail\ conjset = \langle\rangle \\
\quad \textbf{then}\ i\_subRefinement\ ss\ (head\ conjset) \\
\textbf{else} \\
\quad intersect\ (i\_subRefinement\ ss\ (head\ conjset))\ (i\_conjunctionRefinementSet\ ss\ (tail\ conjset))
\end{array}
$$

3.3.2 disjunctionRefinementSet

```
disjunctionRefinementSet = 1*(ws disjunction ws subRefinement)
```

$disjunctionRefinementSet == \text{seq}_1\ subRefinement$

Apply the union operator to the interpretation of each subRefinement

$$
\begin{array}{l}
i\_disjunctionRefinementSet : \\
\quad Substrate \rightarrow disjunctionRefinementSet \rightarrow Sctids\_or\_Error \\
\hline
\forall\ ss : Substrate;\ disjset : disjunctionRefinementSet \bullet \\
i\_disjunctionRefinementSet\ ss\ disjset = \\
\textbf{if}\ tail\ disjset = \langle\rangle \\
\quad \textbf{then}\ i\_subRefinement\ ss\ (head\ disjset) \\
\textbf{else} \\
\quad intersect\ (i\_subRefinement\ ss\ (head\ disjset))\ (i\_disjunctionRefinementSet\ ss\ (tail\ disjset))
\end{array}
$$

3.3.3 subRefinement

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

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

$subRefinement ::=$
$\quad subrefine\_attset\langle\langle attributeSet\rangle\rangle \mid$
$\quad subrefine\_attgroup\langle\langle attributeGroup\rangle\rangle \mid$
$\quad subrefine\_refinement\langle\langle refinement'\rangle\rangle$

$i\_subRefinement :$
$\quad Substrate \to subRefinement \to Sctids\_or\_Error$

$\forall ss : Substrate;\ subrefine : subRefinement \bullet$
$i\_subRefinement\ ss\ subrefine =$
if $subrefine \in \operatorname{ran} subrefine\_attset$
$\quad$ **then** $i\_attributeSet\ ss\ (subrefine\_attset^{\sim} subrefine)$
else if $subrefine \in \operatorname{ran} subrefine\_attgroup$
$\quad$ **then** $i\_attributeGroup\ ss\ (subrefine\_attgroup^{\sim} subrefine)$
else $i\_refinement'\ ss\ (subrefine\_refinement^{\sim} subrefine)$

3.4 attributeSet

attributeSet = subAttributeSet [conjunctionAttributeSet / disjunctionAttributeSet]

$attributeSet == subAttributeSet \times conjunctionOrDisjunctionAttributeSet[0\,..\,1]$
$[attributeSet']$

$conjunctionOrDisjunctionAttributeSet ::=$
$\quad attset\_conjattset\langle\langle conjunctionAttributeSet\rangle\rangle \mid$
$\quad attset\_disjattset\langle\langle disjunctionAttributeSet\rangle\rangle$

$i\_attributeSet :$
$\quad Substrate \to attributeSet \to Sctids\_or\_Error$

$\forall ss : Substrate;\ attset : attributeSet \bullet$
$i\_attributeSet\ ss\ attset =$
(**let** $lhs == i\_subAttributeSet\ ss\ (first\ attset);\ rhs == second\ attset \bullet$
if $\#rhs = 0$
$\quad$ **then** lhs
else if $head\ rhs \in \operatorname{ran} attset\_conjattset$
$\quad$ **then** $intersect\ lhs\ (i\_conjunctionAttributeSet\ ss\ (attset\_conjattset^{\sim}(head\ rhs)))$
else $union\ lhs\ (i\_disjunctionAttributeSet\ ss\ (attset\_conjattset^{\sim}(head\ rhs))))$

$i\_attributeSet' :$
$\quad Substrate \to attributeSet' \to Sctids\_or\_Error$

3.4.1 conjunctionAttributeSet

```
conjunctionAttributeSet = 1*(ws conjunction ws subAttributeSet)
```

$conjunctionAttributeSet == \text{seq}_1\ subAttributeSet$

Apply the intersect operator to the interpretation of each subAttributeSet

$$
\begin{array}{l}
i\_conjunctionAttributeSet : \\
\quad Substrate \rightarrow conjunctionAttributeSet \rightarrow Sctids\_or\_Error \\
\hline
\forall\ ss : Substrate;\ conjset : conjunctionAttributeSet \bullet \\
i\_conjunctionAttributeSet\ ss\ conjset = \\
\textbf{if}\ tail\ conjset = \langle\rangle \\
\quad \textbf{then}\ i\_subAttributeSet\ ss\ (head\ conjset) \\
\textbf{else} \\
\quad intersect\ (i\_subAttributeSet\ ss\ (head\ conjset))\ (i\_conjunctionAttributeSet\ ss\ (tail\ conjset))
\end{array}
$$

3.4.2 disjunctionAttributeSet

```
disjunctionAttributeSet = 1*(ws disjunction ws subAttributeSet)
```

$disjunctionAttributeSet == \text{seq}_1\ subAttributeSet$

Apply the intersect operator to the interpretation of each subAttributeSet

$$
\begin{array}{l}
i\_disjunctionAttributeSet : \\
\quad Substrate \rightarrow disjunctionAttributeSet \rightarrow Sctids\_or\_Error \\
\hline
\forall\ ss : Substrate;\ disjset : disjunctionAttributeSet \bullet \\
i\_disjunctionAttributeSet\ ss\ disjset = \\
\textbf{if}\ tail\ disjset = \langle\rangle \\
\quad \textbf{then}\ i\_subAttributeSet\ ss\ (head\ disjset) \\
\textbf{else} \\
\quad union\ (i\_subAttributeSet\ ss\ (head\ disjset))\ (i\_disjunctionAttributeSet\ ss\ (tail\ disjset))
\end{array}
$$

3.4.3 subAttributeSet

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

$$
\begin{array}{l}
subAttributeSet ::= \\
\quad subaset\_attribute\langle\langle attribute\rangle\rangle\ | \\
\quad subaset\_attset\langle\langle attributeSet'\rangle\rangle
\end{array}
$$

$$
\begin{array}{|l}
\hline
i\_subAttributeSet : \\
\qquad Substrate \rightarrow subAttributeSet \rightarrow Sctids\_or\_Error \\
\hline
\forall\, ss : Substrate;\ subaset : subAttributeSet \bullet \\
i\_subAttributeSet\ ss\ subaset = \\
\textbf{if}\ subaset \in \operatorname{ran} subaset\_attribute \\
\quad \textbf{then}\ i\_attribute\ ss\ (subaset\_attribute^{\sim} subaset) \\
\textbf{else}\ i\_attributeSet'\ ss\ (subaset\_attset^{\sim} subaset) \\
\hline
\end{array}
$$

3.5 attributeGroup

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

$$attributeGroup == cardinality[0\,..\,1] \times attributeSet$$

$$
\begin{array}{|l}
\hline
i\_attributeGroup : Substrate \rightarrow attributeGroup \rightarrow Sctids\_or\_Error \\
\hline
\end{array}
$$

TODO: Finish this

3.6 attribute

The interpretation of an attribute is the place where we transition from a set of *Quads* to a set of *sctIds*. Everything to the right hand side of `cardinality` is treated as quads. *i_cardinality* converts quads to *sctIds*

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

$$
\begin{array}{|l}
\hline
attribute \\
\hline
card : cardinality[0\,..\,1] \\
rf : reverseFlag[0\,..\,1] \\
name : attributeName \\
targets : concreteOrExpressionAttribute \\
\hline
\end{array}
$$

$[reverseFlag]$

$concreteOrExpressionAttribute ::=$
$\quad attrib\_conc\langle\langle concreteAttribute\rangle\rangle \mid$
$\quad attrib\_expr\langle\langle expressionAttribute\rangle\rangle$

$concreteAttribute == concreteComparisonOperator \times concreteValue$

$expressionAttribute == expressionComparisonOperator \times expressionConstraintValue$

$$
\begin{array}{l}
i\_attribute : \\
\quad Substrate \rightarrow attribute \rightarrow Sctids\_or\_Error \\
\hline
\forall\, ss : Substrate;\ att : attribute \bullet \\
i\_attribute\ ss\ att = \\
(\textbf{let}\ att\_sctids == ss.i\_attributeName\ att.name \bullet \\
\textbf{if}\ att\_sctids \in \operatorname{ran} error \\
\quad \textbf{then}\ att\_sctids \\
\textbf{else if}\ att.targets \in \operatorname{ran} attrib\_conc \\
\quad \textbf{then} \\
\quad (\textbf{let}\ conc\_interp == \\
\quad i\_concreteAttribute\ ss\ (result\_sctids\ att\_sctids)\ (attrib\_conc^{\sim}\ att.targets) \bullet \\
\qquad i\_cardinality\ att.card\ conc\_interp) \\
\textbf{else} \\
\quad (\textbf{let}\ exp\_interp == \\
\quad i\_expressionAttribute\ ss\ att.rf\ (result\_sctids\ att\_sctids)\ (attrib\_expr^{\sim}\ att.targets) \bullet \\
\qquad i\_cardinality\ att.card\ exp\_interp))
\end{array}
$$

3.6.1 Attribute Cardinality Interpretation

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

$$
\begin{array}{l}
unlimitedNat ::= num\langle\langle \mathbb{N} \rangle\rangle \mid many \\
cardinality == \mathbb{N} \times unlimitedNat
\end{array}
$$

3.7 Cardinality

This is the interpretation of non-grouped cardinality. It function that takes:

- An optional cardinality
- *Quads_or_Error*: which is one of:
  - A set of *Quad*s and a direction indicator
  - An error

This function:

- Propagates an error condition if one exists
- Returns the empty set if the set of *Quad*s falls outside the cardinality rules
- Returns the set of source *sctIds* in the *Quad*s if the direction is forward (?s attribute t) (*source_direction*)
- Returns the set of target *sctIds* in the *Quad*s if the direction is reverse (s attribute ?t) (*targets_direction*)

$$
\begin{array}{l}
i\_cardinality : \\
\quad cardinality[0\,..\,1] \rightarrow Quads\_or\_Error \rightarrow Sctids\_or\_Error \\
\hline
\forall\, card : cardinality[0\,..\,1];\ qore : Quads\_or\_Error \bullet \\
i\_cardinality\ card\ qore = \\
\textbf{if}\ qore \in \operatorname{ran} qerror \\
\quad \textbf{then}\ error\,(qerror^{\sim} qore) \\
\textbf{else} \\
(\textbf{let}\ qr == evalCardinality[Quad]\ card\,(quads\_for\ qore) \bullet \\
\textbf{if}\ quad\_direction\ qore = source\_direction \\
\quad \textbf{then}\ ok\,\{q : qr \bullet q.s\} \\
\textbf{else}\ ok\,\{q : qr \bullet t\_sctid^{\sim} q.t\})
\end{array}
$$

3.7.1 expressionAttribute

```
expressionConstraintValue = simpleExpressionConstraint / "(" ws (refinedExpressionCon-
straint / compoundExpressionConstraint) ws ")"
```

$$
\begin{array}{l}
expressionConstraintValue ::= \\
\quad expression\_simple\langle\langle simpleExpressionConstraint\rangle\rangle \mid \\
\quad expression\_refined\langle\langle refinedExpressionConstraint'\rangle\rangle \mid \\
\quad expression\_compound\langle\langle compoundExpressionConstraint\rangle\rangle
\end{array}
$$

Expression attribute is a an additional expression we introduced to simplify interpretation. It is the interpretation of:

- An optional `reverseFlag`
- A set of attribute `sctIds`
- An `expressionAttribute` which consists of an `expressionComparisonOperator` and an `expressionConstraintValue`

The interpretation consists of the following steps:

1. Evaluate `expressionConstraintValue`
2. If the evaluation yielded an error propagate it
3. Evaluate the combination of the reverse flag, the attributes, the operator, the result of step 1 and return a set of quads with a direction indicator or an error

$$
\begin{array}{l}
i\_expressionAttribute : \\
\quad Substrate \rightarrow reverseFlag[0\,..\,1] \rightarrow \mathbb{P}\, sctId \rightarrow expressionAttribute \rightarrow Quads\_or\_Error \\
\hline
\forall\, ss : Substrate;\ rf : reverseFlag[0\,..\,1];\ atts : \mathbb{P}\, sctId;\ ea : expressionAttribute \bullet \\
i\_expressionAttribute\ ss\ rf\ atts\ ea = \\
(\textbf{let}\ target\_sctids == \textbf{if}\ (second\ ea) \in \mathrm{ran}\ expression\_simple \\
\quad \textbf{then}\ i\_simpleExpressionConstraint\ ss\ (expression\_simple^{\sim}(second\ ea)) \\
\textbf{else if}\ (second\ ea) \in \mathrm{ran}\ expression\_refined \\
\quad \textbf{then}\ i\_refinedExpressionConstraint'\ ss\ (expression\_refined^{\sim}(second\ ea)) \\
\textbf{else}\ i\_compoundExpressionConstraint\ ss\ (expression\_compound^{\sim}(second\ ea)) \bullet \\
\quad i\_attributeExpressionConstraint\ ss\ rf\ atts\ (first\ ea)\ target\_sctids)
\end{array}
$$

3.7.2 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*

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.

$$
\begin{array}{l}
i\_concreteAttribute : \\
\quad Substrate \rightarrow \mathbb{P}\, sctId \rightarrow concreteAttribute \rightarrow \ Quads\_or\_Error \\
\hline
\forall\, ss : Substrate;\ attids : \mathbb{P}\, sctId;\ ca : concreteAttribute \bullet \\
i\_concreteAttribute\ ss\ attids\ ca = \\
\qquad i\_concreteAttributeConstraint\ ss\ attids\ (first\ ca)\ (second\ ca)
\end{array}
$$

3.8 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

- $[m_1 \,..\, n_1]op[m_2 \,..\, n_2]...$ – 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 sctId'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

$$
\begin{array}{l}
i\_groupCardinality : \\
\qquad Quads\_or\_Error \rightarrow cardinality[0 \,..\, 1] \rightarrow IDGroups \\
\hline
\forall\, quads : Quads\_or\_Error;\ oc : cardinality[0 \,..\, 1];\ uniqueGroups : \mathbb{P}\, groupId; \\
\qquad quadsByGroup : groupId \nrightarrow \mathbb{P}\, Quad \mid \\
\quad uniqueGroups = \{q : quads\_for\ quads \bullet q.g\} \land \\
\quad quadsByGroup = \{g : uniqueGroups;\ q : \mathbb{P}\, Quad \mid \\
\qquad q = \{e : quads\_for\ quads \mid e.g = g\} \bullet g \mapsto (evalCardinality\ oc\ q)\} \bullet \\
i\_groupCardinality\ quads\ oc = \\
\quad id\_groups\, \{sctid : sctId;\ groups : \mathbb{P}\, groupId \mid sctid \in \{q : \bigcup(\mathrm{ran}\ quadsByGroup) \bullet \\
\qquad \textbf{if}\ quad\_direction\ quads = source\_direction\ \textbf{then}\ q.s\ \textbf{else}\ t\_sctid^{\sim} q.t\} \land \\
\quad groups = \{g : \mathrm{dom}\ quadsByGroup \mid (\exists\, q : quadsByGroup\ g \bullet \\
\qquad sctid = \textbf{if}\ quad\_direction\ quads = source\_direction\ \textbf{then}\ q.s\ \textbf{else}\ t\_sctid^{\sim} q.t)\} \bullet \\
\qquad sctid \mapsto groups\}
\end{array}
$$

4 Substrate Interpretations

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

4.1 attributeExpressionConstraint

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

$$expressionComparisonOperator ::= eco\_eq \mid eco\_neq$$

`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.

$$
\begin{array}{l}
i\_attributeExpressionConstraint : \\
\quad Substrate \to reverseFlag[0\,..\,1] \to \mathbb{P}\, sctId \to \\
\quad\quad expressionComparisonOperator \to Sctids\_or\_Error \to Quads\_or\_Error \\
\hline
\forall\, ss : Substrate;\ rf : reverseFlag[0\,..\,1];\ atts : \mathbb{P}\, sctId; \\
\quad op : expressionComparisonOperator;\ subj\_or\_targets : Sctids\_or\_Error \bullet \\
i\_attributeExpressionConstraint\ ss\ rf\ atts\ op\ subj\_or\_targets = \\
\textbf{if}\ subj\_or\_targets \in \operatorname{ran} error \\
\quad \textbf{then}\ qerror\,(error^{\sim} subj\_or\_targets) \\
\textbf{else if}\ \#rf = 0 \land op = eco\_eq\ \textbf{then} \\
\quad quad\_value(\{t : result\_sctids\ subj\_or\_targets; \\
\quad\quad\quad a : atts;\ rels : ss.relationships \mid \\
\quad\quad\quad t\_sctid^{\sim} rels.t = t \land rels.a = a \bullet rels\}, source\_direction) \\
\textbf{else if}\ \#rf = 1 \land op = eco\_eq\ \textbf{then} \\
\quad quad\_value(\{s : result\_sctids\ subj\_or\_targets; \\
\quad\quad\quad a : atts;\ rels : ss.relationships \mid \\
\quad\quad\quad rels.s = s \land rels.t \in \operatorname{ran} t\_sctid \land rels.a = a \bullet rels\}, targets\_direction) \\
\textbf{else if}\ \#rf = 0 \land op = eco\_neq\ \textbf{then} \\
\quad quad\_value(\{t : result\_sctids\ subj\_or\_targets; \\
\quad\quad\quad a : atts;\ rels : ss.relationships \mid \\
\quad\quad\quad t\_sctid^{\sim} rels.t \neq t \land rels.a = a \bullet rels\}, source\_direction) \\
\textbf{else} \\
\quad quad\_value(\{s : result\_sctids\ subj\_or\_targets; \\
\quad\quad\quad a : atts;\ rels : ss.relationships \mid \\
\quad\quad\quad rels.s \neq s \land rels.t \in \operatorname{ran} t\_sctid \land rels.a = a \bullet rels\}, targets\_direction)
\end{array}
$$

4.2 concreteAttributeConstraint

$$
\begin{array}{l}
i\_concreteAttributeConstraint : \\
\quad Substrate \to \mathbb{P}\, sctId \to concreteComparisonOperator \to \\
\quad\quad concreteValue \to Quads\_or\_Error \\
\hline
\forall\, ss : Substrate;\ atts : \mathbb{P}\, sctId;\ op : concreteComparisonOperator; \\
\quad\quad val : concreteValue \bullet \\
i\_concreteAttributeConstraint\ ss\ atts\ op\ val = \\
quad\_value(\{rels : ss.relationships \mid rels.a \in atts \land rels.t \in \operatorname{ran} t\_concrete \land val \in concreteMatch\,(t\_co
\end{array}
$$

$$
\begin{array}{l}
concreteMatch : \\
\quad concreteValue \to concreteComparisonOperator \to \mathbb{P}\, concreteValue
\end{array}
$$

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.

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

4.3 FocusConcept

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

4.3.1 focusConcept

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

focusConcept ::=
focusConcept_m⟨⟨*conceptReference*⟩⟩ |
focusConcept_c⟨⟨*conceptReference*⟩⟩

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

$$
\begin{array}{|l}
\hline
i\_focusConcept : Substrate \rightarrow focusConcept \rightarrow Sctids\_or\_Error \\
\hline
\forall\, ss : Substrate;\ fc : focusConcept \bullet \\
i\_focusConcept\ ss\ fc = \\
\quad \textbf{if}\ focusConcept\_c^{\sim} fc \in conceptReference \\
\quad\quad \textbf{then}\ ss.i\_conceptReference\ (focusConcept\_c^{\sim} fc) \\
\quad \textbf{else}\ i\_memberOf\ ss\ (ss.i\_conceptReference\ (focusConcept\_m^{\sim} fc)) \\
\hline
\end{array}
$$

4.3.2 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*

$$
\begin{array}{|l}
\hline
i\_memberOf : Substrate \rightarrow Sctids\_or\_Error \rightarrow Sctids\_or\_Error \\
\hline
\forall\, ss : Substrate;\ refsetids : Sctids\_or\_Error \bullet \\
i\_memberOf\ ss\ refsetids = \\
\quad \textbf{if}\ refsetids \in \operatorname{ran} error \\
\quad\quad \textbf{then}\ refsetids \\
\quad \textbf{else}\ bigunion\{sctid : result\_sctids\ refsetids \bullet ss.i\_refsetId\ sctid\} \\
\hline
\end{array}
$$

5 Glue and Helper Functions

This section carries various type transformations and error checking functions

5.1 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 *Quad*s or an error condition. If it is a collection *Quad*s, it also carries a direction indicator that determines whether it represents a set of sources or targets.
- **IDGroups** – a map from *sctId*s to the *groupId* they were in when they passed if successful, otherwise an error indication.

$$direction ::= source\_direction \mid targets\_direction$$
$$Quads\_or\_Error ::= quad\_value\langle\langle \mathbb{P}\ Quad \times direction \rangle\rangle \mid qerror\langle\langle ERROR \rangle\rangle$$
$$IDGroups ::= id\_groups\langle\langle sctId \pfun \mathbb{P}\ groupId \rangle\rangle \mid gerror\langle\langle ERROR \rangle\rangle$$

5.2 Result transformations

- **result_sctids** – the set of *sctId*s 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 *groupId* part of in an id group or an empty map if there is error
- **quads_to_idgroups** – convert a set of quads int 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

$$
\begin{array}{l}
result\_sctids : Sctids\_or\_Error \to \mathbb{P}\, sctId \\
quads\_for : Quads\_or\_Error \to \mathbb{P}\, Quad \\
quad\_direction : Quads\_or\_Error \nrightarrow direction \\
to\_idGroups : IDGroups \to sctId \to \mathbb{P}\, groupId \\
quads\_to\_idgroups : Quads\_or\_Error \to IDGroups \\
idgroups\_to\_sctids : IDGroups \to Sctids\_or\_Error \\
quads\_to\_sctids : Quads\_or\_Error \to Sctids\_or\_Error \\
\hline
\forall\, r : Sctids\_or\_Error \bullet result\_sctids\, r = \\
\quad \textbf{if } r \in \operatorname{ran} error \textbf{ then } \emptyset \\
\quad \textbf{else } ok^{\sim} r \\
\\
\forall\, q : Quads\_or\_Error \bullet quads\_for\, q = \\
\quad \textbf{if } qerror^{\sim} q \in ERROR \textbf{ then } \emptyset \\
\quad \textbf{else } first\, (quad\_value^{\sim} q) \\
\\
\forall\, q : Quads\_or\_Error \bullet quad\_direction\, q = \\
\quad second\, (quad\_value^{\sim} q) \\
\\
\forall\, g : IDGroups \bullet to\_idGroups\, g = \\
\quad \textbf{if } gerror^{\sim} g \in ERROR \textbf{ then } \emptyset \\
\quad \textbf{else } id\_groups^{\sim} g \\
\\
\forall\, q : Quads\_or\_Error \bullet quads\_to\_idgroups\, q = \\
\quad \textbf{if } qerror^{\sim} q \in ERROR \textbf{ then } gerror(qerror^{\sim} q) \\
\quad \textbf{else if } quad\_direction\, q = source\_direction \\
\qquad \textbf{then } id\_groups\, \{s : sctId \mid (\exists\, qr : quads\_for\, q \bullet s = qr.s) \bullet \\
\qquad s \mapsto \{qr : quads\_for\, q \bullet qr.g\}\} \\
\quad \textbf{else} \\
\qquad id\_groups\, \{t : sctId \mid (\exists\, qr : quads\_for\, q \bullet t = t\_sctid^{\sim} qr.t) \bullet \\
\qquad t \mapsto \{qr : quads\_for\, q \bullet qr.g\}\} \\
\\
\forall\, g : IDGroups \bullet idgroups\_to\_sctids\, g = \\
\quad \textbf{if } g \in \operatorname{ran} gerror \textbf{ then } ok\, \emptyset \\
\quad \textbf{else } ok\, (\operatorname{dom}(id\_groups^{\sim} g)) \\
\\
\forall\, q : Quads\_or\_Error \bullet quads\_to\_sctids\, q = \\
\quad \textbf{if } q \in \operatorname{ran} qerror \textbf{ then } error\, (qerror^{\sim} q) \\
\quad \textbf{else } ok\, \{qe : quads\_for\, q \bullet qe.s\}
\end{array}
$$

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 and 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.

$$
\begin{array}{l}
\mathit{firstError} : \mathbb{P}\, \mathit{Sctids\_or\_Error} \pfun \mathit{Sctids\_or\_Error} \\
\mathit{qfirstError} : \mathbb{P}\, \mathit{Sctids\_or\_Error} \pfun \mathit{Quads\_or\_Error} \\
\\
\mathit{union}, \mathit{intersect}, \mathit{minus} : \mathit{Sctids\_or\_Error} \fun \mathit{Sctids\_or\_Error} \fun \\
\qquad \mathit{Sctids\_or\_Error} \\
\mathit{bigunion}, \mathit{bigintersect} : \mathbb{P}\, \mathit{Sctids\_or\_Error} \fun \mathit{Sctids\_or\_Error} \\
\hline
\forall\, x, y : \mathit{Sctids\_or\_Error} \bullet \mathit{union}\, x\, y = \\
\qquad \textbf{if } x \in \mathrm{ran}\, \mathit{error} \lor y \in \mathrm{ran}\, \mathit{error} \textbf{ then } \mathit{firstError}\, \{x, y\} \\
\qquad \textbf{else } \mathit{ok}\, ((\mathit{ok}^{\sim} x) \cup (\mathit{ok}^{\sim} y)) \\
\\
\forall\, x, y : \mathit{Sctids\_or\_Error} \bullet \mathit{intersect}\, x\, y = \\
\qquad \textbf{if } x \in \mathrm{ran}\, \mathit{error} \lor y \in \mathrm{ran}\, \mathit{error} \textbf{ then } \mathit{firstError}\, \{x, y\} \\
\qquad \textbf{else } \mathit{ok}\, ((\mathit{ok}^{\sim} x) \cap (\mathit{ok}^{\sim} y)) \\
\\
\forall\, x, y : \mathit{Sctids\_or\_Error} \bullet \mathit{minus}\, x\, y = \\
\qquad \textbf{if } x \in \mathrm{ran}\, \mathit{error} \lor y \in \mathrm{ran}\, \mathit{error} \textbf{ then } \mathit{firstError}\, \{x, y\} \\
\qquad \textbf{else } \mathit{ok}\, ((\mathit{ok}^{\sim} x) \setminus (\mathit{ok}^{\sim} y)) \\
\\
\forall\, \mathit{rs} : \mathbb{P}\, \mathit{Sctids\_or\_Error} \bullet \mathit{bigunion}\, \mathit{rs} = \\
\qquad \textbf{if } \exists\, r : \mathit{rs} \bullet r \in \mathrm{ran}\, \mathit{error} \textbf{ then } \mathit{firstError}\, \mathit{rs} \\
\qquad \textbf{else } \mathit{ok}\, (\bigcup \{r : \mathit{rs} \bullet \mathit{result\_sctids}\, r\}) \\
\\
\forall\, \mathit{rs} : \mathbb{P}\, \mathit{Sctids\_or\_Error} \bullet \mathit{bigintersect}\, \mathit{rs} = \\
\qquad \textbf{if } \exists\, r : \mathit{rs} \bullet r \in \mathrm{ran}\, \mathit{error} \textbf{ then } \mathit{firstError}\, \mathit{rs} \\
\qquad \textbf{else } \mathit{ok}\, (\bigcap \{r : \mathit{rs} \bullet \mathit{result\_sctids}\, r\})
\end{array}
$$

6 Appendix 1 – Optional elements

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

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

7 Appendix 2 – Generic cardinality evaluation

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 $\emptyset$

$[T]$

$$evalCardinality : cardinality[0\,..\,1] \to \mathbb{P}\ T \to \mathbb{P}\ T$$

$$\begin{aligned}
&\forall\ opt_cardinality : cardinality[0\,..\,1];\ t : \mathbb{P}\ T \bullet \\
&evalCardinality\ opt_cardinality\ t = \\
&\quad \textbf{if}\ \#opt_cardinality = 0 \lor \\
&\qquad (\#t \geq first\,(head\ opt_cardinality)) \land \\
&\qquad (second\,(head\ opt_cardinality) = many \lor \\
&\qquad num^{\sim}(second\,(head\ opt_cardinality)) \geq \#t) \\
&\quad \textbf{then}\ t \\
&\quad \textbf{else}\ \emptyset
\end{aligned}$$

8 Appendix 3 - Generic sequence function

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 " $T \times \text{seq}_1\ T$

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 ($\langle\rangle$).

$[T]$

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