A Declarative Semantics for SNOMED CT Expression Constraints

January 22, 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
- **REAL** – a real number
- **STRING** – a string literal
- **GROUP** – a role group identifier
- $\mathbb{N}$ – a non-negative integer
- $\mathbb{Z}$ – an integer

We also introduce several synonyms for *SCTID*:

- **SUBJECT** – a *SCTID* that appears in the *sourceId* position of a relationship.
- **ATTRIBUTE** – a *SCTID* that appears in the *typeId* position of a relationship.
- **OBJECT** – a *SCTID* that appears in the *destinationId* position of a relationship.
- **REFSETID** – a *SCTID* that identifies a reference set

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

$$SUBJECT == SCTID$$
$$ATTRIBUTE == SCTID$$
$$OBJECT == SCTID$$
$$REFSETID == SCTID$$

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 OBJECT or a CONCRETEVALUE

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

2 The Substrate

A substrate represents the context of an interpretation. A substrate consists of:

- **c** 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 this set MUST be treated as an error.**
- **a** The set of *SCTID*s that are considered to be valid attributes in the context of the substrate. **Reference to any *ATTRIBUTE* that is not a member of this set MUST be treated as an error.**
- **r** A set of relationship quads (subject, attribute, target, group)
- **descs** The subsumption (ISA) closure from general to specific (descendants)
- **ancs** The subsumption (ISA) closure from specific to general (ancestors)
- **refset** The reference sets within the context of the substrate whose members are members are concept identifiers (i.e. are in c). 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.

Issue: this is currently the simplest possible kind of Substrate; it represents the DNF form of SNOMED CT with concepts only (i.e., *Expressions* and *Expression Libraries* are not supported).

Issue: this is a *flat* representation (fixed to 1 level of nesting corresponding to role grouping) and does not properly handle Concrete Domains (which almost always involve additional nesting

Open questions: While the Core will not contain cycles, and probably NRC Extensions will not, arbitrary extensions (e.g. as a result of handling Expression libraries) may involve cycles (or at least equivalent concepts).

- **Resolution:** A reflexive, symmetric equivalence relationship (*equiv*) was added to the substrate. How it is determined is left unspecified, but a rule was added saying that an equivalent concept can not be a descendant of its equivalence.

- Is it correct to interpret the transitive closure directly against the substrate relationships (r) set or is there an implication of a reasoner being invoked somewhere, which could potentially render the transitive closure as logically derived from (r). **Answer:** If substrate includes postcoordinated expressions then subsumption would need to be calculated. **Followup:** How does one know whether it includes postcoordinated expressions? How do we express this decision formally?

- Do we want to assert that it is an error to use a non-attribute concept in a attribute position? If not, should we explicitly include a formal definition of the attribute set? **Answer:** Because the ECL doesn't declare this as an error condition, we shouldn't assert in the Z spec. **Resolution:** $a = descs\ linkage\_concept$ assertion pulled from the substrate declaration.

2.1 Substrate Components

Quad Relationships in the substrate are represented a 4 element tuples or "Quads", which consist of a subject, attribute, target and role group identifier.

Quad

$s : SUBJECT$
$a : ATTRIBUTE$
$t : TARGET$
$g : GROUP$

We will also need to recognize some "well known" identifiers: the *is_a* attribute, the *zero_group* and *linkage_concept*, the parent of all attributes

$is\_a : ATTRIBUTE$
$zero\_group : GROUP$
$linkage\_concept : SCTID$

2.2 Substrate

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

1. All subjects and attributes and targets of type *object* in r must be members of the set of concepts, c. All attributes are also members of a.
2. The is_a relation is irreflexive.
3. *childOf* is a function from a concept in c to the set of concepts that are the source of c in an *is_a* relationship in the *zero_group*
4. *parentOf* is a function from a concept c to the set of concepts that are the target of of c in an *is_a* relationship in the *zero_group*
5. *parentOf* and *childOf* are irreflexive.
6. The *equiv* relationship is symmetric and reflexive.

7. The ancestors function, *ancs*, is the *irreflexive* transitive closure of the *childOf* relationship.
8. The descendants function, *descs*, is the *irreflexive* transitive closure of the *parentOf* relationship.
9. The descendants relationship must not contain any concepts that are declared as equivalent to the root.
10. The reference set function is a function from *subset of* c to set of SCTID's in c. A SCTID that is not in the domain of *refset* cannot appear as the target of a *memberOf* function

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Substrate}} \\
c : \mathbb{P}\, \mathit{SCTID} \\
r : \mathbb{P}\, \mathit{Quad} \\
\\
a : \mathbb{P}\, c \\
\mathit{descs} : c \rightarrow \mathbb{P}\, c \\
\mathit{ancs} : c \rightarrow \mathbb{P}\, c \\
\mathit{equiv} : c \leftrightarrow c \\
\mathit{refset} : \mathit{REFSETID} \rightarrow \mathbb{P}\, c \\
\\
\mathit{childOf} : c \rightarrow \mathbb{P}\, c \\
\mathit{parentOf} : c \rightarrow \mathbb{P}\, c \\
\hline
\forall\, \mathit{rel} : r \bullet \mathit{rel}.s \in c \land \mathit{rel}.p \in c \land \mathit{object}\, \mathit{rel}.t \in c \land \mathit{rel}.p \in a \\
\forall\, \mathit{conc} : c;\; g : \mathit{GROUP} \bullet (c, \mathit{is\_a}, c, g) \notin r \\
\\
\forall\, s : c;\; d : \mathbb{P}\, c \bullet \mathit{childOf}\; s = \{ d : c \mid (s, \mathit{is\_a}, d, \mathit{zero\_group}) \in r \} \\
\forall\, d : c;\; s : \mathbb{P}\, c \bullet \mathit{parentOf}\; d = \{ s : c \mid (s, \mathit{is\_a}, d, \mathit{zero\_group}) \in r \} \\
\\
\forall\, x : c \bullet (x \mapsto x) \in \mathit{equiv} \\
\forall\, x_1, x_2 : c \bullet x_1 \mapsto x_2 \in \mathit{equiv} \Leftrightarrow x_2 \mapsto x_1 \in \mathit{equiv} \\
\\
\mathit{descs} = \mathit{childOf}^{+} \\
\mathit{ancs} = \mathit{parentOf}^{+} \\
\forall\, r : \mathit{equiv} \bullet \mathit{first}\; r \in \operatorname{dom} \mathit{descs} \Rightarrow \mathit{second}\; r \notin \mathit{descs}\; r \\
\\
\operatorname{dom} \mathit{refset} \subseteq c \\
\hline
\end{array}
$$

3 Result

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

$$
\begin{aligned}
\mathit{ERROR} ::= {} & \mathit{unknownOperation} \mid \mathit{unknownConceptReference} \mid \\
& \mathit{unknownPredicate} \mid \mathit{unknownRefsetId}
\end{aligned}
$$

$$
\mathit{Result} ::= \mathit{ok}\langle\!\langle \mathbb{P}\, \mathit{SCTID} \rangle\!\rangle \mid \mathit{error}\langle\!\langle \mathit{ERROR} \rangle\!\rangle
$$

4 Interpretation of Intermediate Constructs

This section carries the interpretation of the intermediate constructs – the various forms of expressions and their combinations.

4.1 expressionConstraint

expressionConstraint = (refinedExpressionConstraint / unrefinedExpressionConstraint)

4.1.1 Interpretation

The interpretation of `expressionConstraint` the interpretation of the `refinedExpressionConstraint` or the `unrefinedExpressionConstraint`.

$$
\begin{aligned}
&\mathit{expressionConstraint} ::= \\
&\qquad \mathit{ecrec}\langle\!\langle \mathit{refinedExpressionConstraint} \rangle\!\rangle \mid \\
&\qquad \mathit{ecurec}\langle\!\langle \mathit{unrefinedExpressionConstraint} \rangle\!\rangle
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{i\_expressionConstraint} : \\
&\qquad \mathit{Substrate} \rightarrow \mathit{expressionConstraint} \rightarrow \mathit{Result} \\
\hline
&\forall\, \mathit{ss} : \mathit{Substrate};\ \mathit{ec} : \mathit{expressionConstraint} \bullet \mathit{i\_expressionConstraint}\ \mathit{ss}\ \mathit{ec} = \\
&\quad \textbf{if } \mathit{ecrec}^\sim\, \mathit{ec} \in \mathit{refinedExpressionConstraint} \\
&\qquad \textbf{then } \mathit{i\_refinedExpressionConstraint}\ \mathit{ss}\ (\mathit{ecrec}^\sim\, \mathit{ec}) \\
&\quad \textbf{else if } \mathit{ecurec}^\sim\, \mathit{ec} \in \mathit{unrefinedExpressionConstraint} \\
&\qquad \textbf{then } \mathit{i\_unrefinedExpressionConstraint}\ \mathit{ss}\ (\mathit{ecurec}^\sim\, \mathit{ec}) \\
&\quad \textbf{else } \mathit{error}\ \mathit{unknownOperation}
\end{aligned}
$$

4.1.2 unrefinedExpressionConstraint

An `unrefinedExpressionConstraint` is either a `compoundExpressionConstraint` or a `simpleExpressionConstraint`

```
unrefinedExpressionConstraint =
    compoundExpressionConstraint / simpleExpressionConstraint
```

$$
\begin{aligned}
&\mathit{unrefinedExpressionConstraint} ::= \\
&\qquad \mathit{ucec}\langle\!\langle \mathit{compoundExpressionConstraint} \rangle\!\rangle \mid \\
&\qquad \mathit{usec}\langle\!\langle \mathit{simpleExpressionConstraint} \rangle\!\rangle
\end{aligned}
$$

$$
\begin{array}{|l}
\hline
i\_unrefinedExpressionConstraint : \\
\qquad Substrate \rightarrow unrefinedExpressionConstraint \rightarrow Result \\
\hline
\forall\, ss : Substrate;\ uec : unrefinedExpressionConstraint \bullet \\
i\_unrefinedExpressionConstraint\ ss\ uec = \\
\quad \mathbf{if}\ ucec^{\sim} uec \in compoundExpressionConstraint \\
\qquad \mathbf{then}\ i\_compoundExpressionConstraint\ ss\ (ucec^{\sim} uec) \\
\quad \mathbf{else\ if}\ usec^{\sim} uec \in simpleExpressionConstraint \\
\qquad \mathbf{then}\ i\_simpleExpressionConstraint\ ss\ (usec^{\sim} uec) \\
\quad \mathbf{else}\ error\ unknownOperation \\
\hline
\end{array}
$$

4.1.3 refinedExpressionConstraint

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

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 interpretation of the second option adds no value.

$$
\begin{array}{l}
refinedExpressionConstraint == \\
\qquad unrefinedExpressionConstraint \times refinement
\end{array}
$$

$$
\begin{array}{|l}
\hline
i\_refinedExpressionConstraint : \\
\qquad Substrate \rightarrow\!\!\!\!\!+ refinedExpressionConstraint \rightarrow\!\!\!\!\!+ Result \\
\hline
\forall\, ss : Substrate;\ rec : refinedExpressionConstraint \bullet \\
i\_refinedExpressionConstraint\ ss\ rec = \\
\quad intersect\ (i\_unrefinedExpressionConstraint\ ss\ (first\ rec))(i\_refinement\ ss\ (second\ rec)) \\
\hline
\end{array}
$$

4.2 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 ] focusConcept
```

$$
\begin{array}{l}
constraintOperator ::= descendantOrSelfOf \mid descendantOf \mid \\
\qquad ancestorOrSelfOf \mid ancestorOf \\
simpleExpressionConstraint == constraintOperator[0\mathinner{.\,.}1] \times focusConcept
\end{array}
$$

$$
\begin{array}{|l}
\hline
i\_simpleExpressionConstraint : \\
\qquad Substrate \rightarrow\!\!\!\!\!+ simpleExpressionConstraint \rightarrow\!\!\!\!\!+ Result \\
\hline
\forall ss : Substrate;\ sec : simpleExpressionConstraint \bullet \\
i\_simpleExpressionConstraint\ ss\ sec = \\
\qquad i\_constraintOperator\ ss\ (first\ sec)\ (i\_focusConcept\ ss\ (second\ sec)) \\
\hline
\end{array}
$$

4.3 compoundExpressionConstraint

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

```
compoundExpressionConstraint = conjunctionExpressionConstraint |
       disjunctionExpressionConstraint | exclusionExpressionConstraint |
        "(" compoundExpressionConstraint ")"
```

$$
\begin{array}{l}
compoundExpressionConstraint ::= \\
\qquad cecConj \langle\langle conjunctionExpressionConstraint \rangle\rangle \mid \\
\qquad cecDisj \langle\langle disjunctionExpressionConstraint \rangle\rangle \mid \\
\qquad cecExc \langle\langle exclusionExpressionConstraint \rangle\rangle
\end{array}
$$

$$
\begin{array}{|l}
\hline
i\_compoundExpressionConstraint : \\
\qquad Substrate \rightarrow\!\!\!\!\!+ compoundExpressionConstraint \rightarrow\!\!\!\!\!+ Result \\
\hline
\forall ss : Substrate;\ cec : compoundExpressionConstraint \bullet \\
i\_compoundExpressionConstraint\ ss\ cec = \\
\quad \textbf{if}\ cec^{\sim} cecConj \in conjunctionExpressionConstraint \\
\qquad \textbf{then}\ i\_compoundExpressionConstraint\ ss\ (cecConj^{\sim}\ cec) \\
\quad \textbf{else if}\ cec^{\sim} cecDisj \in disjunctionExpressionConstraint \\
\qquad \textbf{then}\ i\_compoundExpressionConstraint\ ss\ (cecDisj^{\sim}\ cec) \\
\quad \textbf{else if}\ cec^{\sim} cecExc \in exclusionExpressionConstraint \\
\qquad \textbf{then}\ i\_compoundExpressionConstraint\ ss\ (cecExc^{\sim}\ cec) \\
\quad \textbf{else}\ error\ unknownOperation \\
\hline
\end{array}
$$

4.4 conjunctionExpressionConstraint

conjunctionExpressionConstraint is interpreted the conjunction (intersection) of the interpretation of two or more *subExpressionConstraints*

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

$$
conjunctionExpressionConstraint == subExpressionConstraint \times \mathrm{seq}(subExpressionConstraint)
$$

$$
\begin{array}{l}
i\_conjunctionExpressionConstraint : \\
\qquad Substrate \pfun subExpressionConstraint \pfun \mathrm{seq}(subExpressionConstraint) \pfun Result \\
\hline
\forall ss : Substrate;\ sec : subExpressionConstraint;\ secs : \mathrm{seq}(subExpressionConstraint \bullet \\
i\_conjunctionExpressionConstraint\ ss\ sec\ secs = \\
\quad \mathbf{if}\ tail\ secs = \langle\rangle\ \mathbf{then} \\
\qquad intersect\ (i\_subExpressionConstraint\ ss\ sec) \\
\qquad\quad (i\_subExpressionConstraint\ ss\ (head\ secs)) \\
\quad \mathbf{else} \\
\qquad intersect\ (i\_subExpressionConstraint\ ss\ sec) \\
\qquad\quad (i\_conjunctionExpressionConstraint\ ss\ (head\ secs)\ (tail\ secs))
\end{array}
$$

4.5 disjunctionExpressionConstraint

disjunctionExpressionConstraint is interpreted the disjunction (union) of the interpretation of two or more *subExpressionConstraints*

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

$$
\begin{array}{l}
disjunctionExpressionConstraint == \\
\qquad subExpressionConstraint \times \mathrm{seq}(subExpressionConstraint)
\end{array}
$$

$$
\begin{array}{l}
i\_disjunctionExpressionConstraint : \\
\qquad Substrate \pfun subExpressionConstraint \pfun \mathrm{seq}(subExpressionConstraint) \pfun Result \\
\hline
\forall ss : Substrate;\ sec : subExpressionConstraint;\ secs : \mathrm{seq}(subExpressionConstraint \bullet \\
i\_disjunctionExpressionConstraint\ ss\ sec\ secs = \\
\quad \mathbf{if}\ tail\ secs = \langle\rangle\ \mathbf{then} \\
\qquad union\ (i\_subExpressionConstraint\ ss\ sec) \\
\qquad\quad (i\_subExpressionConstraint\ ss\ (head\ secs)) \\
\quad \mathbf{else} \\
\qquad union\ (i\_subExpressionConstraint\ ss\ sec) \\
\qquad\quad (i\_disjunctionExpressionConstraint\ ss\ (head\ secs)\ (tail\ secs))
\end{array}
$$

4.6 exclusionExpressionConstraint

exclusionExpressionConstraint is interpreted the result of removing the members of the second *subExpressionConstraint* from the first

```
exclusionExpressionConstraint =
      subExpressionConstraint exclusion subExpressionConstraint
```

$$exclusionExpressionConstraint == \\ \quad subExpressionConstraint \times subExpressionConstraint$$

$$\begin{array}{|l}
i_exclusionExpressionConstraint : \\
\qquad Substrate \rightarrow\!\!\!\!\!\!+\; subExpressionConstraint \rightarrow\!\!\!\!\!\!+\; subExpressionConstraint \rightarrow\!\!\!\!\!\!+\; Result \\
\hline
\forall\, ss : Substrate;\; sec1, sec2 : subExpressionConstraint \bullet \\
i_exclusionExpressionConstraint\ ss\ sec1\ sec2 = \\
\quad minus\ (i_subExpressionConstraint\ ss\ sec1)\ (i_subExpressionConstraint\ ss\ sec2)
\end{array}$$

4.7 subExpressionConstraint

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

```
subExpressionConstraint =
      simpleExpressionConstraint | "(" compoundExpressionConstraint ")
```

$$\begin{array}{l}
subExpressionConstraint ::= \\
\quad secSec\langle simpleExpressionConstraint\rangle \mid \\
\quad secCec\langle compoundExpressionConstraint\rangle
\end{array}$$

$$\begin{array}{|l}
i_subExpressionConstraint : \\
\qquad Substrate \rightarrow\!\!\!\!\!\!+\; subExpressionConstraint \rightarrow\!\!\!\!\!\!+\; Result \\
\hline
\forall\, ss : Substrate;\; sec : subExpressionConstraint \bullet \\
i_subExpressionConstraint = \\
\quad \textbf{if}\ sec^{\sim} secSec \in simpleExpressionConstraint \\
\qquad \textbf{then}\ i_simpleExpressionConstraint\ ss\ (secSec^{\sim} sec) \\
\quad \textbf{else if}\ sec^{\sim} secCec \in compoundExpressionConstraint \\
\qquad \textbf{then}\ i_compoundExpressionConstraint\ ss\ (secCec^{\sim} sec) \\
\quad \textbf{else}\ error\ unknownOperation
\end{array}$$

4.8 refinement

`refinement` is the interpretation of one or more `attribute` or `attributeGroup`s joined by `compoundOperator`s

```
compoundOperator = conjunction / disjunction / minus
refinement = (attribute / attributeGroup / "(" refinement ")")
         [compoundOperator  refinement]
```

$$\begin{array}{l}
refTarget ::= att\langle\!\langle attribute\rangle\!\rangle \mid attg\langle\!\langle attributeGroup\rangle\!\rangle \\
refinement == refTarget \times \mathrm{seq}(compoundOperator \times refTarget)
\end{array}$$

$$
\begin{array}{l}
i\_refTarget : Substrate \rightarrow refTarget \rightarrow Result \\
i\_refinement : Substrate \rightarrow refinement \rightarrow Result \\
\hline
\forall\, ss : Substrate;\ rt : refTarget \bullet i\_refTarget\ ss\ rt = \\
\quad \textbf{if}\ att^{\sim} rt \in attribute \\
\quad\quad \textbf{then}\ i\_attribute\ ss\ (att^{\sim} rt) \\
\quad \textbf{else if}\ attg^{\sim} rt \in attributeGroup \\
\quad\quad \textbf{then}\ i\_attributeGroup\ ss\ (attg^{\sim} rt) \\
\quad \textbf{else}\ error\ unknownOperation \\
\\
\forall\, ss : Substrate;\ ref : refinement \bullet \\
\quad i\_refinement\ ss\ ref = \\
\quad evalRefinement\ ss\ (i\_refTarget\ ss\ (first\ ref))\ (second\ ref)
\end{array}
$$

$$
\begin{array}{l}
evalRefinement : \\
\quad\quad Substrate \rightarrow Result \rightarrow \mathrm{seq}(compoundOperator \times refTarget) \rightarrow Result \\
\hline
\forall\, ss : Substrate;\ r : Result;\ opt : \mathrm{seq}(compoundOperator \times refTarget) \bullet \\
\quad evalRefinement\ ss\ r\ opt = \\
\quad\quad \textbf{if}\ opt = \langle\rangle \\
\quad\quad\quad \textbf{then}\ r \\
\quad\quad \textbf{else if}\ first\ (head\ opt) = conjunction \\
\quad\quad\quad \textbf{then}\ evalRefinement\ ss\ (intersect\ r (i\_refTarget'\ ss\ (second\ (head\ opt))))\ (tail\ opt) \\
\quad\quad \textbf{else if}\ first\ (head\ opt) = disjunction \\
\quad\quad\quad \textbf{then}\ evalRefinement\ ss\ (union\ r (i\_refTarget'\ ss\ (second\ (head\ opt))))\ (tail\ opt) \\
\quad\quad \textbf{else if}\ first\ (head\ opt) = difference \\
\quad\quad\quad \textbf{then}\ evalRefinement\ ss\ (minus\ r (i\_refTarget'\ ss\ (second\ (head\ opt))))\ (tail\ opt) \\
\quad\quad \textbf{else}\ error\ unknownOperation
\end{array}
$$

$$
i\_refTarget' : Substrate \rightarrow refTarget \rightarrow Result
$$

4.9 expressionConstraintValue

```
expressionConstraintValue = simpleExpressionConstraint /
        "(" (refinedExpressionConstraint /
        compoundExpressionConstraint) ")"
```

$$
\begin{array}{l}
expressionConstraintValue ::= ecvsec\langle\langle simpleExpressionConstraint\rangle\rangle\ | \\
\quad\quad ecvrec\langle\langle refinedExpressionConstraint'\rangle\rangle\ | \\
\quad\quad ecvcec\langle\langle compoundExpressionConstraint\rangle\rangle \\
[refinedExpressionConstraint']
\end{array}
$$

$$
\begin{array}{l}
i\_expressionConstraintValue : Substrate \to \\
\qquad expressionConstraintValue \to Result \\
\hline
\forall ss : Substrate;\ ecv : expressionConstraintValue \bullet \\
i\_expressionConstraintValue\ ss\ ecv = \\
\quad \mathbf{if}\ ecvsec^{\sim}\ ecv \in simpleExpressionConstraint \\
\qquad \mathbf{then}\ i\_simpleExpressionConstraint\ ss\ (ecvsec^{\sim}\ ecv) \\
\quad \mathbf{else\ if}\ ecvrec^{\sim}\ ecv \in refinedExpressionConstraint' \\
\qquad \mathbf{then}\ i\_refinedExpressionConstraint'\ ss\ (ecvrec^{\sim}\ ecv) \\
\quad \mathbf{else\ if}\ ecvcec^{\sim}\ ecv \in compoundExpressionConstraint \\
\qquad \mathbf{then}\ i\_compoundExpressionConstraint\ ss\ (ecvcec^{\sim}\ ecv) \\
\quad \mathbf{else}\ error\ unknownOperation
\end{array}
$$

$$
\begin{array}{l}
i\_refinedExpressionConstraint' : \\
\qquad Substrate \nrightarrow refinedExpressionConstraint' \nrightarrow Result
\end{array}
$$

4.10 Attributes and Attribute Groups

The interpretation of an attribute. `attributeOperator` and `attributeName` determines the set of possible attributes in the substrate relationship table. `reverseFlag` and `expressionConstraintValue` determine the set of candidate targets (if `reverseFlag` is absent) or `subjects` (if `reverseFlag` is present).

`cardinality` determines the minimum and maximum matches. In all cases, only a subset of the subjects (targets if `reverseFlag` is present) in the substrate relationship table will be returned in the interpretation.

```
attribute = [cardinality] [reverseFlag] [attributeOperator] attributeName
       (comparisonOperator concreteValue / "=" expressionConstraintValue )
attributeGroup = "{" attributeSet "}"
attributeSet = attribute [compoundOperator  attributeSet] / "(" attributeSet ")"
cardinality = "[" nonNegativeIntegerValue to (nonNegativeIntegerValue / many) "]"
constraintOperator = descendantOrSelfOf / descendantOf /  ancestorOrSelfOf / ancestorOf
attributeOperator = descendantOrSelfOf / descendantOf
comparisonOperator = "=" / "!" ws "=" /
    ("n"/"N") ("o"/"O") ("t"/"T") ws "=" / "<=" / "<" / ">=" /  ">"
```

4.11 AttributeName and AttributeSubject

i_attributeName verifies that *conceptReference* is a valid concept in the substrate and interprets it as a set of (exactly one) *ATTRIBUTE*

i_attributeSubject interprets the constraintOperator, if any in the context of the attribute name and interprets it as a set of *ATTRIBUTE*s

Question: Should we add any additional validation checks?

$$attributeName == conceptReference$$

$$attributeOperator == \{descendantOrSelfOf, descendantOf\}$$
$$attributeSubject == attributeOperator[0\,..\,1] \times attributeName$$

$$\begin{array}{l}
i_attributeName : Substrate \nrightarrow attributeName \nrightarrow Result \\
i_attributeSubject : Substrate \nrightarrow attributeSubject \nrightarrow Result \\
\hline
\forall ss : Substrate;\ an : attributeName \bullet i_attributeName\ ss\ an = \\
\quad i_conceptReference\ ss\ an \\
\forall ss : Substrate;\ as : attributeSubject \bullet i_attributeSubject\ ss\ as = \\
\quad i_constraintOperator\ ss\ (first\ as)\ (i_attributeName\ ss\ (second\ as))
\end{array}$$

4.12 Attribute

`attribute` consists of an optional `cardinality` and an `attributeConstraint`.

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

$$\begin{array}{l}
\textit{attribute} \\
\hline
card : cardinality[0\,..\,1] \\
attw : attributeConstraint
\end{array}$$

attributeConstraint is either an attribute expression constraint or a concrete value constraint.

$$attributeConstraint ::= aec\langle\langle attributeExpressionConstraint \rangle\rangle \mid$$
$$\quad acvc\langle\langle attributeConcreteValueConstraint \rangle\rangle$$

`attributeExpressionEonstraint` is a combination of a subject constraint (target if reverse flag is true) and an expression constraint value whose interpretation yields a set of SCTID's that filter the target (subject if reverse flag).

$$[reverseFlag]$$

$$\begin{array}{l}
\textit{attributeExpressionConstraint} \\
\hline
a : attributeSubject \\
rf : reverseFlag[0\,..\,1] \\
cs : expressionConstraintValue
\end{array}$$

A concrete value constraint is the combination of a subject constraint and a comparison operator/concrete value whose interpretation yields a set of subject ids. The intersection of the subject and concrete value interpretation is the interpretation of `attributeConcreteValueConstraint`

$$comparisonOperator ::= eq \mid neq \mid gt \mid ge \mid lt \mid le$$

attributeConcreteValueConstraint

$a : attributeSubject$
$op : comparisonOperator$
$v : concreteValue$

The interpretation of an attribute is the interpretation of the cardinality applied against the underlying attribute constraint, producing a set of SCTID's or an error condition

The interpretation of an attribute constraint is the interpretation of either the attribute expression constraint or the concrete expression constraint that produces a set of Quads or an error condition.

The actual interpretations if attribute expression and concrete value are in Section 5

$$
\begin{array}{l}
i\_attribute : \\
\qquad Substrate \pfun attribute \pfun Result \\
i\_attributeConstraint : \\
\qquad Substrate \pfun attributeConstraint \pfun Quads \\
i\_attributeExpressionConstraint : \\
\qquad Substrate \pfun attributeExpressionConstraint \pfun Quads \\
i\_attributeConcreteValueConstraint : \\
\qquad Substrate \pfun attributeConcreteValueConstraint \pfun Quads
\end{array}
$$

$$
\begin{array}{l}
\forall ss : Substrate;\ a : attribute \bullet \\
\quad i\_attribute\ ss\ a = i\_cardinality\ a.card\ (i\_attributeConstraint\ ss\ a.attw) \\[6pt]
\forall ss : Substrate;\ aw : attributeConstraint \bullet i\_attributeConstraint\ ss\ aw = \\
\quad \mathbf{if}\ aec^{\sim} aw \in attributeExpressionConstraint \\
\qquad \mathbf{then}\ i\_attributeExpressionConstraint\ ss\ (aec^{\sim} aw) \\
\quad \mathbf{else\ if}\ acvc^{\sim} aw \in attributeConcreteValueConstraint \\
\qquad \mathbf{then}\ i\_attributeConcreteValueConstraint\ ss\ (acvc^{\sim} aw) \\
\quad \mathbf{else}\ qerror\ unknownOperation \\[6pt]
\forall ss : Substrate;\ aec : attributeExpressionConstraint;\ eca : expressionConstraintArgs \mid \\
\quad eca.atts = i\_attributeSubject\ ss\ aec.p \land \\
\quad eca.rf = aec.rf \land \\
\quad eca.subjOrTarg = i\_expressionConstraintValue\ ss\ aec.cs \bullet \\
\quad i\_attributeExpressionConstraint\ ss\ aec = i\_attributeExpression\ ss\ eca \\[6pt]
\forall ss : Substrate;\ awc : attributeConcreteValueConstraint;\ aca : concreteConstraintArgs \mid \\
\quad aca.atts = i\_attributeSubject\ ss\ awc.p \land \\
\quad aca.op = awc.op \land \\
\quad aca.t = awc.v \bullet \\
\quad i\_attributeConcreteValueConstraint\ ss\ awc = i\_concreteAttributeConstraint\ ss\ aca
\end{array}
$$

4.13 AttributeSet and AttributeGroup

An `attributeGroup` is an `attributeSet`. An `attributeSet` consists of a sequence of one or more attribute constraints joined by `compoundOperators`.

$$
\begin{array}{l}
attributeGroup == attributeSet \\
attributeSet == attribute \times \mathrm{seq}(compoundOperator \times attribute)
\end{array}
$$

$$i\_attributeGroup : Substrate \pfun attributeGroup \pfun Result$$

$$i\_attributeSet : Substrate \pfun attributeSet \pfun Result$$

$$\forall ss : Substrate;\ ag : attributeGroup \bullet i\_attributeGroup\ ss\ ag = i\_attributeSet\ ss\ ag$$

$$\forall ss : Substrate;\ a : attributeSet \bullet i\_attributeSet\ ss\ a = result\ (evalCmpndAtt\ ss\ (i\_groupedAttribute\ ss\ (first\ a))\ (second\ a))$$

4.14 Compound attribute evaluation

The left-to-right evaluation of `attributeSet`. The interpretation takes a *lhs* as a function from *SCTID* to *GROUP* and a *rhs* which is sequence of operator/attribute tuples and recursively interprets the *lhs* and interpretation of the head of the *rhs*.

$$
\begin{array}{l}
evalCmpndAtt : \\
\qquad Substrate \rightarrow IDGroups \rightarrow \mathrm{seq}(compoundOperator \times attribute) \rightarrow IDGroups \\[1ex]
gintersect, gunion, gminus, gfirstError : \\
\qquad IDGroups \rightarrow IDGroups \rightarrow IDGroups \\
\hline
\forall\, ss : Substrate;\ lhs : IDGroups;\ rhs : \mathrm{seq}(compoundOperator \times attribute) \bullet \\
evalCmpndAtt\ ss\ lhs\ rhs = \\
\mathbf{if}\ rhs = \langle\rangle \\
\quad \mathbf{then}\ lhs \\
\mathbf{else\ if}\ first\,(head\ rhs) = conjunction \\
\mathbf{then}\ evalCmpndAtt\ ss\ (gintersect\ lhs(i\_groupedAttribute\ ss\ (second\ (head\ rhs))))\ (tail\ rhs) \\
\mathbf{else\ if}\ first\,(head\ rhs) = disjunction \\
\mathbf{then}\ evalCmpndAtt\ ss\ (gunion\ lhs(i\_groupedAttribute\ ss\ (second\ (head\ rhs))))\ (tail\ rhs) \\
\mathbf{else\ if}\ first\,(head\ rhs) = difference \\
\mathbf{then}\ evalCmpndAtt\ ss\ (gminus\ lhs(i\_groupedAttribute\ ss\ (second\ (head\ rhs))))\ (tail\ rhs) \\
\mathbf{else}\ gerror\ unknownOperation \\[1ex]
\forall\, a, b, r : IDGroups \mid \\
\quad r = \mathbf{if}\ gerror^{\sim} a \in ERROR \lor gerror^{\sim} b \in ERROR\ \mathbf{then}\ gfirstError\ a\ b \\
\quad \mathbf{else}\ gv\ (gv^{\sim} a \cap gv^{\sim} b) \bullet \\
\quad gintersect\ a\ b = r \\[1ex]
\forall\, a, b, r : IDGroups \mid \\
\quad r = \mathbf{if}\ gerror^{\sim} a \in ERROR \lor gerror^{\sim} b \in ERROR\ \mathbf{then}\ gfirstError\ a\ b \\
\quad \mathbf{else}\ gv\ (gv^{\sim} a \cup gv^{\sim} b) \bullet \\
\quad gunion\ a\ b = r \\[1ex]
\forall\, a, b, r : IDGroups \mid \\
\quad r = \mathbf{if}\ gerror^{\sim} a \in ERROR \lor gerror^{\sim} b \in ERROR\ \mathbf{then}\ gfirstError\ a\ b \\
\quad \mathbf{else}\ gv\ (gv^{\sim} a \setminus gv^{\sim} b) \bullet \\
\quad gminus\ a\ b = r
\end{array}
$$

$$
\begin{array}{l}
i\_groupedAttribute : \\
\qquad Substrate \rightarrow attribute \rightarrow IDGroups \\
\hline
\forall\, ss : Substrate;\ a : attribute \bullet \\
\quad i\_groupedAttribute\ ss\ a = i\_groupCardinality\ (i\_attributeConstraint\ ss\ a.attw)\ a.card
\end{array}
$$

4.15 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: the *gresult* function seems to express what is described below more simply

$$
\begin{array}{l}
i\_groupCardinality : \\
\qquad Quads \rightarrow cardinality[0..1] \rightarrow IDGroups \\
\hline
\forall\, quads : Quads;\ oc : cardinality[0..1];\ uniqueGroups : \mathbb{P}\, GROUP; \\
\qquad quadsByGroup : GROUP \nrightarrow \mathbb{P}\, Quad \mid \\
\quad uniqueGroups = \{q : qids\ quads \bullet q.g\} \wedge \\
\quad quadsByGroup = \{g : uniqueGroups;\ q : \mathbb{P}\, Quad \mid \\
\qquad q = \{e : qids\ quads \mid e.g = g\} \bullet g \mapsto (evalCardinality\ oc\ q)\} \bullet \\
i\_groupCardinality\ quads\ oc = \\
\quad gv\ \{sctid : SCTID;\ groups : \mathbb{P}\, GROUP \mid sctid \in \{q : \bigcup(\mathrm{ran}\ quadsByGroup) \bullet \\
\qquad \textbf{if}\ qidd\ quads = subjects\ \textbf{then}\ q.s\ \textbf{else}\ q.t\} \wedge \\
\quad groups = \{g : \mathrm{dom}\ quadsByGroup \mid (\exists\, q : quadsByGroup\ g \bullet \\
\qquad sctid = \textbf{if}\ qidd\ quads = subjects\ \textbf{then}\ q.s\ \textbf{else}\ q.t)\} \bullet \\
\qquad sctid \mapsto groups\}
\end{array}
$$

4.16 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 subjects / targets of the set of quads
3. Otherwise return an empty set

$$
\begin{array}{l}
i\_cardinality : \\
\qquad cardinality[0..1] \rightarrow Quads \rightarrow Result \\
\hline
\forall\, oc : cardinality[0..1];\ qr : Quads \bullet \\
i\_cardinality\ oc\ qr = \\
\quad \textbf{if}\ qerror^{\sim}\, qr \in ERROR\ \textbf{then}\ result\ (gresult\ qr) \\
\quad \textbf{else}\ result\ (gresult\ (qv\ (evalCardinality\ oc\ (qids\ qr), qidd\ qr)))
\end{array}
$$

evalCardinality Evaluate the cardinality of a set, returning the set if it meets the constraints, otherwise return the empty set.

$$evalCardinality : cardinality[0\,..\,1] \rightarrow \mathbb{P}\,Quad \rightarrow \mathbb{P}\,Quad$$

$$\begin{array}{l}
\forall\, oc : cardinality[0\,..\,1];\ s : \mathbb{P}\,Quad \bullet evalCardinality\ oc\ s = \\
\quad \mathbf{if}\ \#oc = 0\ \mathbf{then}\ s \\
\quad \mathbf{else\ if}\ (\#s \geq first\,(head\ oc)) \land \\
\qquad (second\,(head\ oc) = many \lor num^{\sim}(second\,(head\ oc)) \leq \#s) \\
\quad \mathbf{then}\ s \\
\quad \mathbf{else}\ \emptyset
\end{array}$$

5 Substrate Interpretations

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

5.1 AttributeExpressionConstraint

expressionConstraintArgs carries the arguments necessary to evaluate an attribute expression

- **rf** – If present, return subjects matching attribute/target subjects
- **subjOrTarg** – Set of subject or target SCTIDS (or error)
- **atts** – Set of attributes to evaluate

expressionConstraintArgs

$$\begin{array}{l}
rf : reverseFlag[0\,..\,1] \\
subjOrTarg : Result \\
atts : Result
\end{array}$$

Evaluate an attribute expression, returning the set of quads in the substrate relationship table with the supplied subject / attribute if *rf* is absent and attribute / target if *rf* is present. The interpretation returns an error there is an error in either the subject/targets or attributes.

$$
\begin{array}{l}
i\_attributeExpression : \\
\qquad Substrate \rightarrow expressionConstraintArgs \rightarrow Quads \\
\hline
\forall\, ss : Substrate;\ args : expressionConstraintArgs; \\
\quad sti : \mathbb{P}\, TARGET;\ atts : \mathbb{P}\, ATTRIBUTE \mid \\
sti = v^{\sim} args.subjOrTarg \land atts = v^{\sim} args.atts \bullet \\
i\_attributeExpression\ ss\ args = \\
\quad \mathbf{if}\ error^{\sim}(bigunion\{args.subjOrTarg, args.atts\}) \in ERROR \\
\quad \mathbf{then} \\
\qquad qfirstError\{args.subjOrTarg, args.atts\} \\
\quad \mathbf{else\ if}\ \neg\ (sti \cup atts) \subseteq ss.c \\
\quad \mathbf{then} \\
\qquad qerror\ unknownConceptReference \\
\quad \mathbf{else\ if}\ \#args.rf = 0 \\
\quad \mathbf{then} \\
\qquad qv(\{s : SUBJECT;\ t : sti;\ a : atts;\ g : GROUP;\ re : ss.r \mid \\
\qquad\quad re.t = t \land re.p = a \bullet re\}, subjects) \\
\quad \mathbf{else} \\
\qquad qv(\{s : sti;\ a : atts;\ t : TARGET;\ g : GROUP;\ re : ss.r \mid \\
\qquad\quad re.s = t \land re.p = a \bullet re\}, targets)
\end{array}
$$

5.2 ConcreteAttributeConstraint

```
concreteValue =  QM stringValue QM / "#" numericValue
stringValue = 1*(anyNonEscapedChar / escapedChar)
numericValue = decimalValue / integerValue
integerValue = ( ["-"/"+"] digitNonZero *digit ) / zero
```

The mechanism for interpreting *concreteValue* is not fully specified at this point. Much of the rest of the machinery is focused around interpreting constraints in the context of quads - subject, attribute, target and group, so the function is currently defined as returning a *Quads*, but another type or interpretation may be in order.

$$concreteValue ::= stringValue \mid integerValue \mid realValue$$

concreteConstraintArgs The arguments to the concrete value function.

- **atts** – list of attributes to test
- **op** – operator
- **t** – value to test against

Note: *reverseFlag* has no meaning for concrete constraints.

$$
\begin{array}{l}
\underline{\ concreteConstraintArgs\ } \\
atts : Result \\
op : comparisonOperator \\
t : concreteValue
\end{array}
$$

$$
\begin{array}{l}
i\_concreteAttributeConstraint : \\
\qquad Substrate \rightarrow concreteConstraintArgs \rightarrow Quads
\end{array}
$$

5.3 ConstraintOperator

Issue: What is the behavior on a transitive closure that has a cycle? At the moment we always remove the focus concept on *descendantsOf* and *ancestorsOf*

Interpretation: Apply the substrate forward (*descs*) or reverse (*ancs*) functions to a set of SCTID's in the supplied *Result*. Error conditions are propagated.

$$
\begin{array}{l}
i\_constraintOperator : \\
\qquad Substrate \nrightarrow constraintOperator[0\,..\,1] \nrightarrow Result \nrightarrow Result \\
completeFun : (SCTID \nrightarrow \mathbb{P}\, SCTID) \rightarrow SCTID \rightarrow \mathbb{P}\, SCTID \\
\hline
\forall\, ss : Substrate;\ oco : constraintOperator[0\,..\,1];\ refset : Result \bullet \\
i\_constraintOperator\ ss\ oco\ refset = \\
\quad \mathbf{if}\ error^{\sim} refset \in ERROR \lor \#oco = 0 \\
\qquad \mathbf{then}\ refset \\
\quad \mathbf{else\ if}\ head\ oco = descendantOrSelfOf \\
\qquad \mathbf{then}\ v(\bigcup\{id : ids\ refset \bullet completeFun\ ss.descs\ id \cup \{id\}\}) \\
\quad \mathbf{else\ if}\ head\ oco = descendantOf \\
\qquad \mathbf{then}\ v(\bigcup\{id : ids\ refset \bullet completeFun\ ss.descs\ id \setminus \{id\}\}) \\
\quad \mathbf{else\ if}\ head\ oco = ancestorOrSelfOf \\
\qquad \mathbf{then}\ v(\bigcup\{id : ids\ refset \bullet completeFun\ ss.ancs\ id \cup \{id\}\}) \\
\quad \mathbf{else\ if}\ head\ oco = ancestorOf \\
\qquad \mathbf{then}\ v(\bigcup\{id : ids\ refset \bullet completeFun\ ss.ancs\ id \setminus \{id\}\}) \\
\quad \mathbf{else}\ error\ unknownOperation \\
\forall\, f : (SCTID \nrightarrow \mathbb{P}\, SCTID);\ id : SCTID \bullet completeFun\ f\ id = \\
\quad \mathbf{if}\ id \in \mathrm{dom}\, f\ \mathbf{then}\ f\ id\ \mathbf{else}\ \emptyset
\end{array}
$$

5.4 FocusConcept

```
focusConcept = conceptReference / (memberOf focusConcept)
```

Issue: The recursive `memberOf` operator should be replaced with two options:

- One deep – return all the direct descendants of the focus concept
- Transitive Closure – return the transitive closure of the recursive evaluation of the concept

Interpretation:

1. *conceptReference* – *i_conceptReference conceptReference*

2. *memberOf conceptReference* – apply the substrate *vs* function if the SCTID of conceptReference is in the domain of the function, otherwise return an error.
3. *closure conceptReference* – apply the substrate *tcvs* function if the SCTID of conceptReference is in the domain of the function, otherwise return an error

$$valueSetOptions ::= memberOf \mid closure$$
$$focusConcept \mathrel{\widehat=} [opts : valueSetOptions[0 \mathinner{..} 1];\ cr : conceptReference]$$

$$\begin{array}{l}
i_focusConcept : Substrate \rightarrow focusConcept \rightarrow Result \\
\hline
\forall\, ss : Substrate;\ fc : focusConcept \bullet \\
i_focusConcept\ ss\ fc = \\
\quad \textbf{if}\ \#fc.opts = 0\ \textbf{then}\ i_conceptReference\ ss\ fc.cr \\
\quad \textbf{else}\ (\textbf{let}\ sctid == (toSCTID\ fc.cr) \bullet \\
\qquad \textbf{if}\ sctid \notin \operatorname{dom} ss.vs\ \textbf{then}\ error\ unknownConceptReference \\
\qquad \textbf{else if}\ head\ fc.opts = memberOf\ \textbf{then}\ v\ (ss.vs\ sctid) \\
\qquad \textbf{else}\ v\ (ss.tcvs\ sctid))
\end{array}$$

5.5 ConceptReference

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

Interpretation: conceptReference is interpreted as SCTID if it is in the list of concepts in the substrate, otherwise as an error.

$$[conceptReference]$$

$$\begin{array}{l}
toSCTID : conceptReference \rightarrow SCTID \\
i_conceptReference : Substrate \rightarrow conceptReference \rightarrow Result \\
\hline
\forall\, ss : Substrate;\ c : conceptReference \bullet i_conceptReference\ ss\ c = \\
\quad \textbf{if}\ (toSCTID\ c) \in ss.c\ \textbf{then}\ v\{(toSCTID\ c)\} \\
\quad \textbf{else}\ error\ unknownConceptReference
\end{array}$$

6 Helper Functions

- *Quads* is either a collection of *Quad*s or an error condition. If *Quad*s, it also carries a direction indicator that determines whether it represents a set of subjects or targets.
- *IDGroups* is a map from SCTID's to their passing groups or an error condition

$direction ::= subjects \mid targets$
$Quads ::= qv \langle\!\langle \mathbb{P}\, Quad \times direction \rangle\!\rangle \mid qerror \langle\!\langle ERROR \rangle\!\rangle$

$IDGroups ::= gv \langle\!\langle SCTID \pfun \mathbb{P}\, GROUP \rangle\!\rangle \mid gerror \langle\!\langle ERROR \rangle\!\rangle$

Results functions

- *ids* – return the set of sctids in a result or the empty set if there is an error
- *qids* – return the set of quads in a quads result or an empty set if there is an error
- *qidd* – return the direction of a quads result. Undefined if error
- *gids* – return the sctid to group map in an id group or an empty map if there is error
- *gresult* – convert a set of quads int a set of id groups
- *result* – convert a set of id groups into a simple result.

$$
\begin{array}{l}
ids : Result \rightarrow \mathbb{P}\, SCTID \\
qids : Quads \rightarrow \mathbb{P}\, Quad \\
qidd : Quads \nrightarrow direction \\
gids : IDGroups \rightarrow SCTID \rightarrow \mathbb{P}\, GROUP \\
gresult : Quads \rightarrow IDGroups \\
result : IDGroups \rightarrow Result
\end{array}
$$

$$
\begin{array}{l}
\forall\, r : Result \bullet ids\ r = \\
\quad \mathbf{if}\ error^{\sim} r \in ERROR\ \mathbf{then}\ \emptyset \\
\quad \mathbf{else}\ v^{\sim} r \\[6pt]
\forall\, q : Quads \bullet qids\ q = \\
\quad \mathbf{if}\ qerror^{\sim} q \in ERROR\ \mathbf{then}\ \emptyset \\
\quad \mathbf{else}\ first\,(qv^{\sim} q) \\[6pt]
\forall\, q : Quads \bullet qidd\ q = \\
\quad second\,(qv^{\sim} q) \\[6pt]
\forall\, g : IDGroups \bullet gids\ g = \\
\quad \mathbf{if}\ gerror^{\sim} g \in ERROR\ \mathbf{then}\ \emptyset \\
\quad \mathbf{else}\ gv^{\sim} g \\[6pt]
\forall\, q : Quads \bullet gresult\ q = \\
\quad \mathbf{if}\ qerror^{\sim} q \in ERROR\ \mathbf{then}\ gerror(qerror^{\sim} q) \\
\quad \mathbf{else\ if}\ qidd\ q = subjects \\
\qquad \mathbf{then}\ gv\,\{s : SCTID \mid (\exists\, qr : qids\ q \bullet s = qr.s) \bullet \\
\qquad s \mapsto \{qr : qids\ q \bullet qr.g\}\} \\
\quad \mathbf{else} \\
\qquad gv\,\{t : SCTID \mid (\exists\, qr : qids\ q \bullet t = qr.t) \bullet \\
\qquad t \mapsto \{qr : qids\ q \bullet qr.g\}\} \\[6pt]
\forall\, g : IDGroups \bullet result\ g = \\
\quad \mathbf{if}\ gerror^{\sim} g \in ERROR\ \mathbf{then}\ error(gerror^{\sim} g) \\
\quad \mathbf{else}\ v(\mathrm{dom}(gv^{\sim} g))
\end{array}
$$

Definition of the various functions that are performed on the result type. *firstError* is not fully defined – it takes a set of results with one or more errors and returns an aggregation of all of the errors.

$$
\begin{array}{|l}
\hline
\mathit{union}, \mathit{intersect}, \mathit{minus} : \mathit{Result} \rightarrow \mathit{Result} \rightarrow \mathit{Result} \\
\mathit{bigunion}, \mathit{bigintersect} : \mathbb{P}\, \mathit{Result} \rightarrow \mathit{Result} \\
\mathit{firstError} : \mathbb{P}\, \mathit{Result} \nrightarrow \mathit{Result} \\
\mathit{qfirstError} : \mathbb{P}\, \mathit{Result} \nrightarrow \mathit{Quads} \\
\hline
\forall\, x, y : \mathit{Result} \bullet \mathit{union}\; x\; y = \\
\quad \textbf{if } \mathit{error}^{\sim} x \in \mathit{ERROR} \textbf{ then } x \\
\quad \textbf{else if } \mathit{error}^{\sim} y \in \mathit{ERROR} \textbf{ then } y \\
\quad \textbf{else } v\,((v^{\sim} x) \cup (v^{\sim} y)) \\
\forall\, x, y : \mathit{Result} \bullet \mathit{intersect}\; x\; y = \\
\quad \textbf{if } \mathit{error}^{\sim} x \in \mathit{ERROR} \textbf{ then } x \\
\quad \textbf{else if } \mathit{error}^{\sim} y \in \mathit{ERROR} \textbf{ then } y \\
\quad \textbf{else } v\,((v^{\sim} x) \cap (v^{\sim} y)) \\
\forall\, x, y : \mathit{Result} \bullet \mathit{minus}\; x\; y = \\
\quad \textbf{if } \mathit{error}^{\sim} x \in \mathit{ERROR} \textbf{ then } x \\
\quad \textbf{else if } \mathit{error}^{\sim} y \in \mathit{ERROR} \textbf{ then } y \\
\quad \textbf{else } v\,((v^{\sim} x) \setminus (v^{\sim} y)) \\
\forall\, \mathit{refset} : \mathbb{P}\, \mathit{Result} \bullet \mathit{bigunion}\; \mathit{refset} = \\
\quad \textbf{if } \exists\, r : \mathit{refset} \bullet \mathit{error}^{\sim} r \in \mathit{ERROR} \textbf{ then } \mathit{firstError}\; \mathit{refset} \\
\quad \textbf{else } v\,(\bigcup\{r : \mathit{refset} \bullet \mathit{ids}\; r\}) \\
\forall\, \mathit{refset} : \mathbb{P}\, \mathit{Result} \bullet \mathit{bigintersect}\; \mathit{refset} = \\
\quad \textbf{if } \exists\, r : \mathit{refset} \bullet \mathit{error}^{\sim} r \in \mathit{ERROR} \textbf{ then } \mathit{firstError}\; \mathit{refset} \\
\quad \textbf{else } v\,(\bigcap\{r : \mathit{refset} \bullet \mathit{ids}\; r\}) \\
\hline
\end{array}
$$

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.

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