4
Functional Layer of Exploration Framework

The goal of this chapter is to formally describe exploration processes in terms of an expressive set of operations. The formalization is, therefore, the basis for further discussions on exploration strategies, expressivity evaluations, and interface/interaction issues, as established in the framework of reference presented in chapter 3. First, we present our generic data model, which we use for defining the operations. Next, we describe the subjects involved in the exploration process. Finally, we describe each exploration action as an operation applied to the items of the data model and the exploration process as a composition of these operations.

4.1.Preliminary notations

This section describes some basic notation that we use extensively along the chapter.

Sets can be denoted by both the enumeration of its elements, e.g. S = {e_1, e_2, …, e_n}, and by specifying a property *P* of its members:

$$S=\{e|P(e)\}$$

The properties are described in terms of a set of logical operators:

→ stands for “implies”

↔ stands for “if and only if”

∧ stands for “and”

∨ stands for “or”

¬ stands for “not”

∃ stands for “there exists”

∀ stands for “for all”

≡ stands for “equivalent of”

- **Empty Set**: we use the symbol ∅ for denoting empty sets.
- **Ordered Pairs:** we delimit ordered pairs using “<” “>”, e.g. <a1,a2>, <i1,i2>.

- **Cartesian Products:** in order to specify graphs, i.e., sets of ordered pairs, we use the Cartesian product symbol "×" between the sets. As an example: let A= {i1, i2} and B={a1, a2}. The Cartesian product $A \times B$ is:

$$A \times B = \{< i1, a1 >, < i1, a2 >, < i2, a1 >, < i2, a2 >\}$$

- **Multiple Cartesian Products:** In order to represent n Cartesian products of a set we use exponents:

$$A \times A \times A = A^3$$

- **Binary Relations:** let A and B be any two sets, we define a binary relation $\Phi \subseteq A \times B$ as a subset of their Cartesian product. We also denote an element $< i, j > \in \Phi$ as $i \Phi j$.
- **Cardinality:** we denote |S| as the number of members of a set S:
  - $|\{i_1, i_2\}| = 2$
  - $|\{i_1, i_2, i_3\}| = 3$
- **Indexing sets:** sometimes it is necessary to refer to specific items in a partially ordered set. We use the following notation in these cases. Let $S = \{i_1, i_2, i_3\}$ be a partially ordered set:
  - $S_{[1]} = i_1$
  - $S_{[2]} = i_2$
  - $S_{[3]} = i_3$
- **Set operations:** let A and B be any two sets, we establish the following operations over sets:
  - Union:

$$A \cup B = \{x | x \in A \vee x \in B\}$$

  - Intersection:

$$A \cap B = \{x | x \in A \wedge x \in B\}$$

  - Difference:

$$A - B = \{x | x \in A \wedge \neg x \in B\}$$

4.2.The Exploration Process

The exploration process is normally approached as a set of interdependent states (FERRÉ; HERMANN, 2012; TZITZIKAS; MANOLIS, 2016), where each state consists of two components: *Intention* and *Extension*. The *Intention* is a description in some language of the desired set of items in the state. The

Extension is the actual set of items corresponding to the intention. As an example, consider a keyword search for items matching the keywords "Semantic Web". The intention is the keyword expression and the extension is the set of matched items. In this work, we define an exploration process $Po = <St, Dep>$ as a set of exploration states *St* and a state dependency relation $Dep: St \times St$, such that, each pair of states $< s1, s2 > \in Dep$ represents a relation between a state and its subsequent state. The application of an operation to *s1* leads the explorer to the state *s2*. Each state $sx \in St$ is an *invocation* of an exploration operator, which is defined as the set of parameter attributions for the execution of the operator. A formal description of the operation invocations can be found in Definition 15.

Since each state is generated by the use of an exploration operation, we can also describe the exploration process as a functional composition of the operations. Let $Opr = \{op_1, op_2, ..., op_n\}$ be a set of exploration operations. The exploration process *Po* is described by:

$Po = op_n(op_{n-1}(...(op_1(args_1), args_2), args_{n-1}), args_n)$, *where args is a list of arguments specific for each operation*

4.3.Data Model

There are many data models described in the literature, such as the relational model, RDF, and a variety of NoSQL models. Although the validity of these models has been extensively proven for their respective domains of problems, we see the exploration process as not attached to the specificities of a single model; it can be described in terms of a generic data model that can be further mapped to each one of these. Therefore, we devised a simplified version of the Entity-Relationship model (CHEN, 1976), which suffices for our purposes.

Independently of how the data is represented, the user always manipulates items and relationships among them. Items can be organized in groups, such as papers by author or papers by venue. Groups can also be formed along more than one dimension, such as papers by author by publication year. Therefore, the design solution adopted was to model items and relations as nested relations. As an example, "papers by author by publication year" is a three-level nesting, having the papers grouped by year inside a group for each author. Nesting relations can be represented as trees, as Figure 10 shows. We call "exploration set" any nested relation generated by the execution of an exploration action.

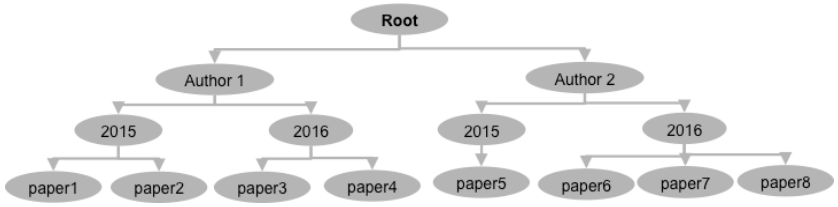

Figure 10 - Nesting of papers by author by publication year.

This data model is very similar to the one used in NoSQL document-oriented databases (CATTELL, 2011), where the nestings of an exploration set can be mapped to collections and nested arrays.

4.3.1.Dataset, Items, and Relations

Definition 1: Dataset

A dataset $D = <I, R>$ is described in terms of a set of exploration items I and a set of trees R, where each tree $t \in R$ represents a set of relationships between exploration items. A tree is defined as an ordered pair $T= < t, \preccurlyeq_T >$, where $t \subseteq I$ is a set of exploration items and $\preccurlyeq_T \subseteq I \times I$ is a partial-order relation that is reflexive, transitive, and antisymetric.

The binary relation $\preccurlyeq_T$ defines the child-parent relationships of the tree and is also well founded, i.e., there is a least item $i \in t$ that has no parent, which is the *root* of the tree. More formally, the properties of the relation trees are the following:

- Reflexive: $\forall i \in t \; i \preccurlyeq_T i$
- Antisymetric: $\forall i, j \in t \; (i \preccurlyeq_T \; j \wedge j \preccurlyeq_T i) \rightarrow i = j$
- Transitive: $\forall i, j, k \in t \; (i \preccurlyeq_T \; j \wedge j \preccurlyeq_T k) \rightarrow \; i \preccurlyeq_T k$
- Well-founded: $\exists k \in t \; \forall i \in t \; < i, k > \notin \preccurlyeq_T$

Since trees are binary relations, we denote trees as a set of relationships in the form *T={<item₁, item₂>, ..., <itemₙ₋₁, itemₙ>}*, e.g., *T={<root, p2>, <p2, a1>, <p2, a2>, <root, p3>, <p3, a2>,<p3, a3>}*. We can also shorten this notation by representing pairs having the first element in common in the following way:

T={<root,{<p2, {a1, a2}>, <p3, {a2, a3}>}>}

Definition 2: Parent and Children of items

Let $T =< t, \preccurlyeq_T >$ be a tree. Let *<itemᵢ, itemⱼ>* be an edge in $\preccurlyeq_T$, the parent of the *itemⱼ* is denoted as *p(itemⱼ) = itemᵢ*.

The children of an item c($item_i$) is the set items having $item_i$ as parent:

$$c(item_i) = \{item_j | < item_i, item_j > \in \preccurlyeq_T\}$$

Let *S={<ids, <a1, {<p1, {f1, f2}>}>>}*. Consider the following examples:

p(f1) = p1

p(p1) = a1

p(a1) = root(s) = ids

p(ids)= null

c(f1) = ∅

c(p1) = {f1, f2}

c(a1) = {p1}

Definition 3: Levels of a tree

Let $T =< t, \preccurlyeq_T >$ be a relation tree and $i \in t$ be an exploration item. The predecessors of *i* is defined as:

$$pred(i) = \{y | y \preccurlyeq_T i\}$$

We define the height of *i* as the cardinality of the set of its predecessors:

$$h(i) = |pred(i)|$$

The nth level of a tree is defined by the set of items having the height *n*:

$$L_n(T) = \{x \in t | h(x) = n\}$$

The set of leaves of *T* is defined as the set of all elements $x \in t$ that has no child items:

$$lf(s) = \{x | c(x) = \emptyset\}$$

Definition 4: Relations

Each relation $r \in R$, in the dataset <I, R>, is represented as a tree having the relation identifier as its root. We refer to relation sets by specifying their ID preceded by ":". As an example, Figure 11 shows the relation between publications and their respective authors.

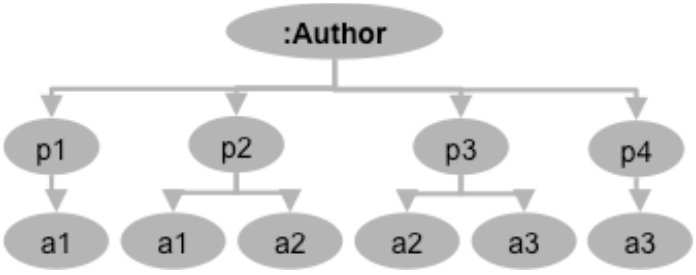

Figure 11 - Relation tree representing publication-author relationships

Using the tree notation, the author relation can also be denoted as follows:

$$Author = \{<:Author, \{<p2, \{a1, a2\}>, <p3, \{a2, a3\}>\}>\}$$

For conciseness purposes, we sometimes omit the root and denote relations in the following way:

$$:Author = \{<p2, \{a1, a2\}>, <p3, \{a2, a3\}>\}$$

Definition 5: Domain and Image of Relations

Each relation $Rt \in R$ has two properties: *domain(Rt)* and *image(Rt)*. *domain(Rt)* is the set of members of the domain of *Rt*, i.e., the set of all items that are children of the root item:

$$domain(Rt) = \{i | i \in c(root(Rt))\}$$

The *image(Rt)* property is the set of all second items of the relationships having the domain items in the first position:

$$image(Rt) = \{j | i \in domain(Rt) \wedge < i, j > \in \preccurlyeq_{Rt}\}$$

As an example, let *:Author = {<p1, {a1, a2}>, <p2, {a2, a3}>}* be a relation between publications and authors. The domain and image sets of *:Author* are:

$$domain(:Author) = \{p1, p2\}$$
$$image(:Author) = \{a1, a2, a3\}$$

The restricted image of a relation *Rt* on a domain item *domItem* is composed of all items of the image of *Rt* that has *domItem* as its domain:

$$image(Rt, domItem) = \{j | < domItem, j > \in \preccurlyeq_{Rt}\}$$

Consider the following restricted image example over the *:Author* relation:

$$image(:Author, p1) = \{a1, a2\}$$

Similarly, the restricted domain of a relation is defined as the set of all items related to a specific image item in *Rt*:

$$domain(Rt, imgItem) = \{j | < j, imgItem > \in \preccurlyeq_{Rt}\}$$

Consider the following restricted domain example:

$$domain(:Author, a2) = \{p1, p2\}$$

In order to keep the notation concise, we denote restricted images and domains using "[]" as follows:

image(:Author, p1) ≡ :Author[p1];

domain(:Author, a2) ≡ :Author-1[a2]

Definition 6: Join

Two relations can be joined in the following way:

$$RJoin(R1, R2) = \{< i, j > \mid < i, k > \in \preccurlyeq_{R1} \wedge \exists k < k, j > \in \preccurlyeq_{R2}\}$$

As an example, let *R1 = {<p1, a1>, <p2, a2>}* and *R2 = {<a1, f1>, <a2, f2>}*, the join between these two relations is defined by:

$$RJoin(R1, R2) = \{< p1, f1 >, < p2, f2 >\}$$

In order to join more than two relations, we make join compositions. Suppose a third relation *R3 = {<f1, i1>, <f2, i2>}*. The join between *R1*, *R2*, and *R3* is:

$$RJoin(RJoin(R1, R2), R3) = \{< p1, i1 >, < p2, i2 >\}$$

Definition 7: Relation Path

A relation path $R_1, \ldots, R_n$ is an ordered set of relations with non-empty join:

$$RJoin(\ldots(RJoin(R_1, R_2)\ \ldots), R_n) \neq \emptyset$$

We denote relation paths by concatenating their identifiers. For example, suppose a relation *:Author = {<p1, a1>, <p2, a2>}* and a relation *:Affiliation = {<a1, f1>, <a2, f2>}*. We denote the path formed by these two relations as:

:Author:Affiliation

The same functions domain and image (Definition 5) can also be applied to relation paths:

:Author:Affiliation^{-1}[f1] ≡ domain(:Author:Affiliation, f1) = {p1}

:Author:Affiliation[p1] ≡ image(:Author:Affiliation, p1) = {f1}

4.3.2.Exploration Sets and Exploration Items

Definition 8: Exploration Set

An exploration set is a tree $T =< t, \preccurlyeq_T >$ generated by the application of an exploration operation to a previous tree, where the starting point is always the dataset. Since exploration sets are relation trees, the domain, image, restricted domain, and restricted image functions (Definition 5) also applies to them.

As an example of an exploration set, consider the result set of a grouping operation carried out over the following sets and relations.

- A set of publications *T = {<Pub, p1>, <Pub, p2>, <Pub, p3>, <Pub, p4>}*, identified as "Pub";
- A set of authors *A = {<ath, a1>, <ath, a2>}*, idenfied as "ath".
- A schema relation between publications and authors:
  :Author = {<p1, a1>, <p2, {a1, a2}>, <p3, {a2, a3}>,<p4, a3>}

The application of a grouping function to the set *T* will result in the following exploration set:

$$\{<rsid, \{<a1, \{p1, p2\}>, <a2, \{p2, p3, p4\}>\}>\} \leftarrow Group(T,:Author)$$

In the example above, "rsid" stands for the exploration set identifier.

Definition 9: Path Sets

A *path set* of an exploration set $S =< s, \preccurlyeq_S >$ is the set of predecessor sets of the leaves of *S*:

$$paths(S) = \bigcup_{l \in lf(s)} \{pred(l) \cup \{l\}\}$$

As an example, consider the set *S = {<ids, <a1, {<p1, {f1, f2}>}>>}*, the paths of *S* are:

$$paths(S) = \{\{ids, a1, p1, f1\}, \{ids, a1, p1, f2\}\}$$

Each path in *paths(S)* is a sequence in the form $\{a_i\}_{i=1}^{h(l)}$ ordered by the relation $\preccurlyeq_S$, where, *h(l)* is the height of the leaf item *l*, which is the greatest element of the path.

Definition 10: heads and tails of branches

Let *p = {idp, a1, p1, f1}* be a path of a set *S*, we define the head of *p* as the item directly connected to the root of the tree: *head(p) = a1*. The tail of *p* is the item that has no child: *tail(p) = f1*.

Definition 11: Root Replacement

Let Ph be a path set (Definition 9). The root replacement function *r(Ph,nroot) = Ph'* maps a path set *Ph* onto another path set *Ph'* having the root element (least element) replaced by *nroot*. This function is used for copying a path set from one set to another. Consider the following definition:

$$r(Ph, nroot) = \{< item_i, item_j > \in Ph \mid \forall < p_i, item_j > \in Ph\ (p_i = root(Ph) \rightarrow item_i = nroot) \wedge (p_i \neq root(P) \rightarrow item_i = p_i)$$

As an example, let *Ph ={ids, f1, f2}* be a path set. The copy of the path *Ph* to the root *rt* is defined as:

r(Ph, rt) = {rt, f1, f2}, where all relations with *ids* were replaced by *rt* preserving the ordering of the path set.

4.4.A Model of Exploration Operations

Here we describe the set of operations comprehensive enough to at least describe the state-of-the-art exploration tools currently proposed in the literature, with regards to data manipulation. First, we present the notation used for functions in our framework. Next, we describe each operation in terms of a general description, a signature, a formal description of their results, and usage examples.

4.4.1.Notational Convention for Functions

Although the framework is concerned with exploration functions, i.e. functions that cause state transitions in the exploration process, some auxiliary and domain specific functions can be used as arguments of exploration functions. We denote auxiliary functions using the following rule:

$$functionName(arg_1, arg_2, ..., arg_n)$$

Exploration functions are distinguished from auxiliary functions by having the first character of the name capitalized and having the input state identifier specified before the function name as follows:

$$StateId.ExplorationFunctionName(arg_1, arg_2, ..., arg_n)$$

Consider the following examples. Let *Pb = {<pb, {p1, p2, p3}>}* be an exploration set of publications and *:Author = {<p1, a1>, <p2, a1>, <p3, a2>}* be the relation between the publications and their authors. The refinement operation of publications having *a1* as author is as follows:

$$Rs \leftarrow Pb.Refine(equals(:Author[\%item], a1))$$

In this notation, *Refine* is the exploration function applied to the set *Pb*. *Refine* receives the auxiliary predicate function *equals* as argument that will be evaluated for all publications in *Pb*. The predicate *equals* receives the restricted image of *:Author* on each publication of *Pb*, represented as the parameter *%item*. The application of the operation is preceded by the attribution symbol "←" and the identifier of the result set "Rs". For conciseness purposes we omit the *%item* parameter in some cases:

$$equals(:Author[\%item], a1) \equiv equals(:Author, a1)$$

4.4.2.Extension-Oriented Operations

Extension-oriented operations are mappings between exploration sets. The operations are defined as follows:

Unite

Description: the union operation receives two exploration sets and unites their path sets.

Signature: *Unite(A, B): R × R → R*

Formal definition: let *rs* be the root item of the result set. The union operation is defined as:

$$Unite(A, B) = r(paths(A), rs) \cup r(paths(B), rs)$$

The union operation is defined as the union between the paths of *A* and *B*, mapped to a common root item *rs*. Consider the following examples of the *Unite* operation:

Example 1: let *A = {<sa, {p1, p2, p3}>}* and *B = {<sb, {p3, p4, p5}>}* be two exploration sets containing hypothetical publications. The union between *A* and *B* is:

$$\{<rs,\{ p1, p2, p3, p4, p5\}>\} \leftarrow Unite(A, B)$$

Example 2: let *A* = *{<sa, {<a1, {p1, p2, p3}>, <a2, {p3, p4}>}>}* and *B* = *{<sb, {<a2, {p5, p6}>, <a3, {p8, p9}>}>}* be two sets of publications grouped by author. The union between *A* and *B* is:

$$\{<rs, \{<a1, \{p1, p2, p3\}>, <a2, \{p3, p4, p5, p6 \}>, <a3, \{p8, p9\}>\}>\} \leftarrow Unite(A,B)$$

Intersect

Description: the intersect operation computes the intersection between the paths of the input sets.

Signature: *Intersect(A, B): R × R → R*

Formal Definition: let *A* and *B* be two input sets. Let *rs* be the root of the result set. The intersection between *A* and *B* is defined as:

$$Intersect(A, B) = r(paths(A), rs) \cap r(paths(B), rs)$$

The set of paths of the result set is the intersection between the paths of *A* and *B* with the root replaced by the *rs* item. Consider the following examples:

Example 1: let *A = {<sa, {p1, p2, p3}>}* and *B ={<sb, {p2, p3, p5}>}* be two sets of hypothetical publications. The intersection between *A* and *B* is:

$$\{<rs, \{p2, p3\}>\} \leftarrow Intersect\ (A, B)$$

Example 2: Intersecting two sets of publications grouped by author: let *A = {<sa, {<a1, {p1, p2, p3}>, <a2, {p3, p4}>}>}* and *B = {<sb, {<a1, {p2, p3, p5}>, <a2, {p3, p5, p6}>,<a3, p8>}>}* be two grouped sets:

$$\{<rs, \{<a1, \{p2, p3\}>, <a2, p3>\}>\} \leftarrow Intersect\ (A, B)$$

Diff

Description: the difference function computes the difference between the path sets of the input sets.

Signature: *Diff(A, B): R × R → R*

Formal Description: let *rs* be the root item of the result set. The difference between *A* and *B* is defined as follows:

$$Diff(A,B) = r(paths(A), rs) - r(paths(B), rs)$$

The paths in the result set are all paths in *A* that do not appear in *B* under the same root *rs*. Consider the following examples:

Example 1: let *A = {<sa, {p1, p2, p3}>}* and *B = {<sb, {p2, p3, p5}>}* be two publication sets. The difference between *A* and *B* is:

$$\{<rs, p1>\} \leftarrow Diff(A, B)$$

Example2: let *A = {<sa,{<a1, {p1, p2, p3}>, <a2, {p3, p4}>}>}* and *B = {<sb, {<a1, {p2, p3, p5}>, <a2, {p3, p5, p6}>, <a3, p8>}>}* be two sets of publications grouped by author. The difference between *A* and *B* is:

$$\{<rs,\{<a1, p1>,<a2, p4>\}>\} \leftarrow Diff(A, B)$$

Pivot

Description: maps the leaf items of the input exploration set onto another set of related items.

Signature: *Pivot(A, Relations):* $R \times R^n \rightarrow R$, where R^n is the set of all relation paths of size *n*.

Formal Description: let *rs* be the root item of the result set of a pivot operation. We define *Pivot* as follows:

$$Pivot(A, Relations) = \bigcup_{imgItem \in Relations[lf(A)]} < rs, imgItem >$$

The result set is, therefore, the set of all relation images rooted by *rs*. Let *T = {<st, {p1, p2, p3}>}* be a set of scientific publications, *A = {<sa, {a1, a2, a3}>}* be a set of authors, and *F = {<sf, {f1, f2, f3}>}* be a set of authors' affiliations. Let *:Author = {<p1, a1>, <p2, {a1, a2}>, <p3, a3>}* be a relation between publications and authors. Let *:Affiliation = {<a1, f1>, <a2, f2>, <a3, f3>}* be a relation between authors and affiliations. Consider the following examples:

Example 1: pivoting from publications to authors:

$$\{<rs, \{a1, a2, a3\}>\} \leftarrow T.Pivot(:Author)$$

Example 2: pivoting from publications to authors' affiliations through a property path:

$$\{<rs, \{f1, f2, f3\}>\} \leftarrow T.Pivot(:Author:Affiliation)$$

Refine

Description: filters an exploration set using filtering functions (Definition 12) and path patterns (Definition 13) defined by the user.

Definition 12: Filtering Function

A filter $F: I \rightarrow \{true, false\}$ is a predicate that maps an exploration item onto a Boolean value that indicates whether the item must be filtered or not. For the following definitions, let $C = \{filter_1, filter_2, \ldots, filter_n\}$ be the set of filtering functions available for explorations.

Definition 13: Path Pattern

Let $S = < s, \preccurlyeq_s >$ be an exploration set. A path pattern $T = <F, E>$ is a set of filters $F \subseteq C$ and a set of edges between filters, such that, for each $< u, v > \in E$ there is a mapping $< \sigma(u), \sigma(v) > \in \preccurlyeq_s$. Figure 12 shows an exploration set and an example of a path pattern of filters. The mapping $\sigma(filter): C \rightarrow I$ matches the nodes in the path having the same height of the nodes in *E:*

$$\sigma = \{< u, j > | u \in F \wedge j \in s \wedge h(u) = h(j)\}$$

Let *True* be a special filter that always evaluates to *true*. The root of the path pattern is always *True*, therefore, for every pattern, <*True, v*> is an edge in *E* and there is a mapping <*True, root(S)*> $\in \sigma$.

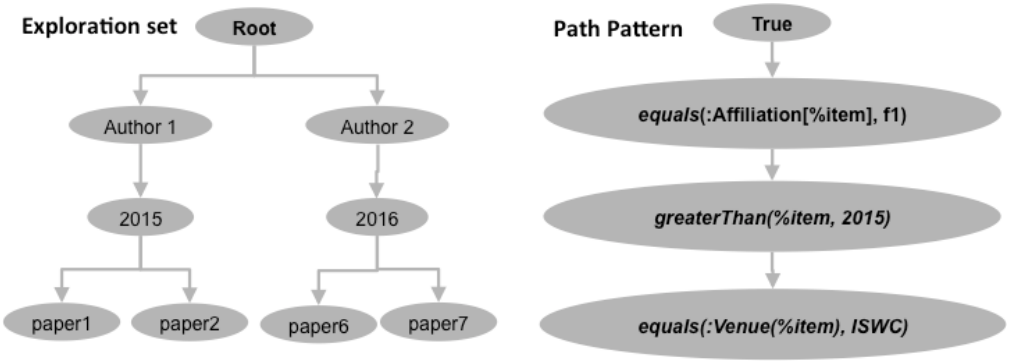

Figure 12 - An exploration set and a path pattern containing the filters for each level.

The path pattern of Figure 12 matches each level of the set, where authors are filtered by the *:Affiliation* relation, the years are filtered by the predicate *greaterThan(%item, 2015)*, and the papers are filtered by venue. This definition is an adaptation of the graph patterns presented in (AGGARWAL; WANG, 2010, ch. 3). Having defined the structure of the path patterns, we can now define the *Refine* operation:

Signature: *Refine(A, PathPattern): R×(C×C) → R*, where *(*C×*C)* defines a binary relation of filters, which represents a path pattern.

Formal Definition: the refine operation is defined as follows:

$$Refine(A, PathPattern) = \{path \in paths(A) \mid \forall < f1, f2 > \in PathPattern \; \exists < \sigma(f1), \sigma(f2) > \in path \wedge f1\big(\sigma(f1)\big) \wedge f2\big(\sigma(f2)\big)\}$$

The definition above filters every path in which, for all pairs of filters *<f1,f2>* in the *PathPattern*, there is a matching edge in the path and the filters *f1* and *f2* hold for the matched nodes.

The expressivity of the refine operation is strongly related to the filtering functions and compositions allowed by the tool. The following functions are usually found in exploration tools:

- *equals(item, value2)*: *I×I→{true,false}*: tests the equality of an exploration item with a specific value;
- *equals(item, Relation, value)*: *I×R×I→{true,false}:* tests if a related value of an exploration item, defined by relation, is equal to value;
- *matchAll(item, keywordPattern)*: *I×L^n→{true,false}:* tests if an exploration item matches all keywords in a keyword pattern. Keyword patterns can be represented as a *n*-tuple of literals in a set *L*;
- *matchOne(item, keywordPattern)*: *I×L^n→{true,false}* tests if an exploration item matches at least one of the keywords in a keyword pattern;
- *not(f)*: *{true,false} →{true,false}* negates another filtering function.

Examples:

Let *T = {<st, {p1, p2, p3}>}* be set of scientific publications, *A = {<sa, {a1, a2, a3}>}*, and *F = {<sf, {f1, f2, f3}>}* be a set of authors' affiliations. Let *:Author = {<p1, a1>, <p2, a1>, <p2, a2>, <p3, a3>}* be a relation between publications

and authors. Let *:Affiliation = {<a1, f1>, <a2, f2>, <a3, f3>}* be a relation between authors and affiliations. Consider the following examples for *Refine*:

Example 1: find papers whose author is a1

{<rs, {p1, p2}>} ← T.Refine(<True, equals(:Author, a1)>)

Example 2: find papers whose authors are *a1* and *a2*:

```
{<rs, p1>} ← Intersect(
        T.Refine(<True, equals(:Author, a1 )>),
        T.Refine(<True, equals(:Author, a2)>)
)
```

Example 3: find papers whose authors are *a1* or *a3*:

```
{<rs, {p1 , p3}>} ← Unite(
        T.Refine(<True, equals(:Author, a1 )>),
        T.Refine(<True, equals(:Author, a3)>)
)
```

Example 4: find papers whose authors' affiliations are equals to *f2* or *f3*:

```
{<rs, {p2 , p3}>} ← Unite(
        T.Refine(<True, equals(:Author:Affiliation, f2)>),
        T.Refine(<True, equals(:Author:Affiliation, f3)>)
)
```

Example 5: refining sets with more than two levels: let *S = {<ss, {<p1, a1>, <p2, {a1, a2}>, <p3, a3>}>}* be a relation between papers and authors. The refinement is expressed as follows:

{<rs, {<p1, a1>,<p2, a1>}>} ← S.Refine(<True,True,equals(a1)>)

Example 6: refining by parent: it is possible to filter items by the parent items using the *p(item)* function (Definition 17):

```
{<rs,{<p1, a1>,<p2, {a1, a2}>}>} ← Unite(
    S.Refine(<True, True, equals(p(%item), p1)>),
    S.Refine(<True, True, equals(p(%item), p2)>)
```

)

In order to simplify the notation of path patterns, for the remainder of the chapter we define the patterns as pairs specifying the position and the filter to be applied at this position. We assume that the remaining positions contain the *True* filter. Therefore, the filter of the Example 6 can be summarized as <3,*equals(p(%item), p2)*>. In case of filters applied only to the last level, we omit the position. Therefore, for Example 6 the following equivalence holds:

$$<True, True, equals(p(\%item), p1)> \equiv <3, equals(p(\%item), p2) \equiv equals(p(\%item), p2)$$

Group

Description: groups an exploration set based on a grouping function, which defines the group of each item. The *Group* operation creates nestings in the result set.

Definition 14: grouping function

The grouping relation *gR: I* × *I* is a relation between an item of the dataset and its grouping item. The rationale in formalizing *gR* as a relation, and not a function, is to allow items to be mapped to more than one group when necessary. Next, we define the grouping operation:

Signature: *Group(A, gR): R* × *Gf* → *R*, where *Gf* is the set of all grouping relations available in the exploration environment.

Formal Definition: let *rs* be the root item of the result set. The *Group* operation is defined as follows:

$$Group(A, gR) = \bigcup_{path \in paths(A)} \bigcup_{i \in gR[tail(path)]} r(path \cup \{< p(tail(path)), i >, < i, tail(path) >\} - < p(tail(path)), tail(path) >, rs)$$

In the formal definition, the grouping item $i \in gR[tail(path)]$ adds another level to the input set tree, where the parent of the tail is related to the item *i* and *i* is related to the tail item of the path. However, the tail item will have two parents

after this operation, this breaks the condition of a tree. For this reason, we remove the relation between the tail and its previous parent in the last part of equation.

Examples:

Let *T={<st, {p1, p2, p3, p4}>}* be set of scientific publications, *A={<sa,{a1,a2}>}*, and *F={<sf, {f1,f2,f3}>}* be a set of authors' affiliations. Let *:Author={<p1,a1>,<p2,a1>,<p3,a2>,<p2,a2>,<p3,a3>,<p4,a3>}* be a relation between publications and authors. Let *:Affiliation={<a1,f1>,<a2,f1> <a3,f2>}* be a relation between authors and affiliations.

Example 1: grouping *T* by author:

$$\{<rs,\{<a1,\{p1,p2\}>,<a2,\{p2,p3\}>,<a3,\{p3,p4\}>\}>\}\leftarrow T.Group(:Author)$$

Example 2: grouping *T* by property path *:Author:Affiliation*:

$$\{<rs,\{<f1,\{p1,p2\ p3\}>,<f2,\{p3,p4\}>\}>\} \leftarrow T.GroupBy(:Author:Affiliation)$$

Example 3: Grouping a grouped set:

$$\{<rs,\{<f1,\{<a1,\{p1,p2\}>,<a2,p3>\}>,<f2,<a3,\{p3,p4\}>>\}>\} \leftarrow T.GroupBy(:Author).GroupBy(:Affiliation)$$

Example 4: grouping by a computed relation: let *:Title={<p1,t1>,<p2,t2>,<p3,t3>,<p4,t4>,<p5,t5>}* be a relation between publications and their respective titles. Let *glv(p):T→T* be a function that maps publications to the publications that has the most similar title using the *Levenschtein* string similarity method. Grouping publications by title similarity is represented as:

{<rs,{<p2, {p1, p2, p4}>,<p3, {p3, p5}>}>}← T.GroupBy{|p| glv(%item)}, where the grouping element (domain) is the publication whose title distance is closest among all grouped publications (centroid).

Rank

Description: the *Rank* operation ranks the paths of the input set given a score function applied to the paths' items.

Signature: *Rank(A, level, scoreFunction): R×N×Sf→R×R*, where *level* is an integer indicating the level containing the items to be scored, and *Sf* is the set of score functions available.

Formal description: the result set has the same items and structure of the input set, respecting the ordering relation chosen for ranking. Let *rs* be the root item of the result set, and *scr(item): I→N* be a score function available in the environment. Let *lv* be the level of the tree containing the items to be scored. The ranking is given by:

$$Rank(A, lv, scr) = \{< r(path_i, rs), r(path_j, rs) > | \forall path_i, path_j \in paths(A) < path_i, path_j > \leftrightarrow scr(path_i[lv]) \geq scr(path_j[lv])\}$$

The *Rank* operation establishes an ordering relation over the set of paths of the input set based on the score of the ranking items, given by $scr(path_i[lv])$.

Examples:

Let *T={<st, {p1, p2, p3, p4}>}* be set of scientific publications and *:Year={<p2,2001>,<p1,2002>,<p3,2003>,<p4,2004>}* be a set of publication years.

Example 1: rank by relation image in descending order:

{<rs, {p4, p3, p1, p2}>} ← T.Rank(1, :Year[%item])

Example 2: rank by relation image in ascending order:

*{<rs, {p2, p1, p3, p4}>} ← T.Rank(1, :Year[%item] * -1)*

Example 3: ranking sets with *height = 2*. Let *G = {<sg,{<a1,{p2, p1}>,<a2, {p3, p4}>}>}* be a group of publications by author. Ranking the last level by publication year:

{<rs,{<a1, {p1, p2}>,<a2, {p4, p3}>}>} ← G.Rank(2, :Year[%item])

Example 4: ranking groups of items. Let *J= {<sj,{<j1,{p5,p6}>,<j2,{p7,p8}>}>}* be a group of publications by journal release, and *:Release = {<j1, 2002>, <j2, 2004>}* be the relation between a journal and its release year. Ranking the groups by journals' release years is expressed as:

{<rs,{<j2, {p7, p8 }>, <j1, {p5, p6}>}>} ← J.Rank(1, :Release[%journal]), where 1 denotes the level contains the ranking subjects.

Map

Description: the *Map* operation applies a function to each item of a given input set level or to each relationship of the input set paths. *Map* can be used to aggregated values, such as counts and sums, generate transformation sets, and combinations of items/relations.

Signature: *Map(A, lv, f): R×N×Mr → R*, where *Mr* is the set of all mapping functions available. The mapping function *f* can be classified as a *Transformation*, *Aggregation*, and *Combination* function. The structure of the result set depends on the class of the mapping function.

The *Map* operator can be *horizontal* or *vertical*. The horizontal mapping applies the mapping function to the children set of each item in the level specified by the *lv* parameter, denoted as $L_{lv}(A)$ as Figure 13 shows.

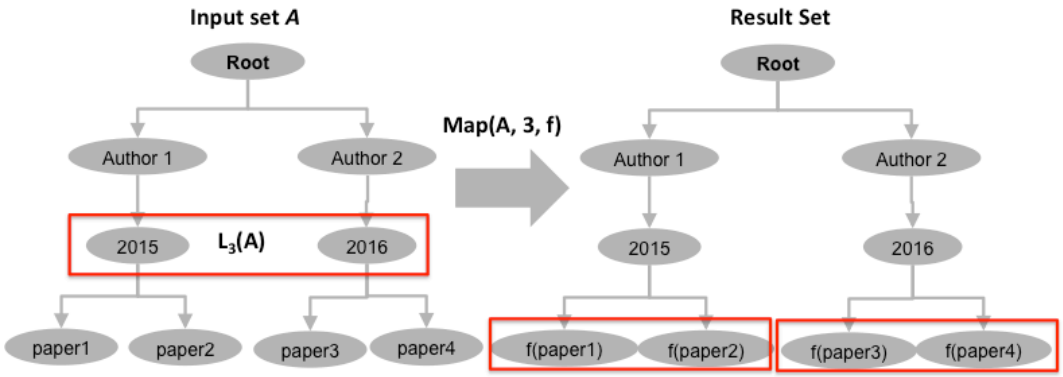

Figure 13 - Representation of a horizontal mapping in the third level

In the horizontal mapping of Figure 13 the mapping level is the next to last and the mapping function *f* is applied to the items of the children sets *c(2015)* and *c(2016)*. The applications of *f* are isolated from each children set. Next, we define the classes of mapping functions that are commonly found in tabular exploration tools as horizontal maps.

Transformation Map

Transformation functions are instances of the original higher-order map function usually found in functional programming languages (BIRD; WADLER,

1988 ch. 3), such as Ruby and Python, where the function *f* is applied to each item of the children set. The transformation *Map* is defined as:

$$Map(A, lv, f) = \{< item, c(item) > | item \in L_{lv}(A) \longrightarrow c(item) =$$

$$\{f(c(item)_{[1]}), f(\, c(item)_{[2]}), \dots, f\big(c(item)_{[n]}\big)\}\}$$

The definition above applies *f* to each children set of each item in the level *lv* of the input set, represented as $L_{lv}(A)$. The structure of the input set is preserved, except for the mapping level, which is replaced by the results of *f*. Consider the following example:

> **Example 1:** let *M = {<sm, {150.00, 160.50, 135.73}>}* be a set of book prices in *US$*. The user can apply a transformation mapping to get the values in *R$*. Suppose the conversion function *rs(value) = value * 3,50*.

$$\{<rs, \{525.00, 561.75, 475.05\}>\} \leftarrow M.Map(1, rs(\%item))$$

Aggregation Map

The aggregation function reduces the set of children of each item in the level *lv* to a single value, as defined by the following equation:

$$Map(A, lv, f) = \{< item, c(item) > | item \in L_{lv}(A) \longrightarrow c(item)$$

$$= \{f(c(item)_{[1]}, f(c(item)_{[2]}, f\big(\dots f\big(c(item)_{[n]}, null\big) \dots\big)\}\}$$

In the definition above, the function *f* is applied to each child item where the next application is the input of the previous application. As an example, for an item having two children, the application of *f* is $f(child1, f(child2, null))$. The *null* value is an initial value that will be combined with the next applications. This structure allows, for example, the computation of counts where each application adds one unit to the results of the previous applications, having the *null* value as 0 (zero). The following examples illustrates aggregation maps:

> **Example 2:** counting elements: let *P = {<sp, {p1, p2, p3}>}* be a set of publications. *count: I* $\longrightarrow$ *N* be an aggregator function that maps a set of items onto their number of elements. The expression bellow counts the number of items in *P*:
>
> $$\{<rs, 3>\} \leftarrow P.Map(1, count(\%item))$$
>
> **Example 3:** mapping sets with multiple levels: let *Y = {<sy,{<2005,{p1, p2 ,p3 ,p4}>,<2006,{p5, p6, p7}>}>}* be the set of

publications grouped by publication year. In order to map this group onto counts by year, we do:

$$\{<rs,\{<2005, 4>,<2006, 3>\}>\} \leftarrow Y.Map(2, count(\%item))$$

The structure of the aggregation maps is an adaptation of the higher-order function *fold* in functional programming (BIRD; WADLER, 1988 ch. 3)

Combination Map

The combination map applies a n-ary function $f(\text{item}_1,\ldots,\text{item}_n)$ to combinations of size n of the children set of an item. Let f be a combination function and lv be the level to map. Let n be both the arity of f and the size of the combination, and $c(item)^n$ be n Cartesian products of the children set. The combinational map is defined as:

$$Map(A, lv, f) = \{< item, c(item) > | item \in L_{lv}(A) \longrightarrow c(item) = \bigcup_{<i_1,\ldots i_n> \in c(item)^n} f(i_1, \ldots i_n) >\}$$

Although the definition above comprises n Cartesian products of the children set $c(item)^n$, this function is usually applied to a subset of $c(item)^n$. For example, Tableau and SeCo tools allow the user to map items to values that are combinations of two or more attributes. In these tools, for example, the user can map a set of *Orders* having the columns {*productId*, *clientId, amount, individualPrice*} to a column *Total = amount*individualPrice*. Considering the children set of each *order* as the total set of values for all columns, the application of the map function is restricted to a subset $Cs \subset c(order)^2$ where the children are values for the attributes *amount* and *individualPrice*.

A variation of the *Map* operation is the *Vertical Map,* where the mapping function is applied to each edge of each path from the root to the leaves.

Vertical Map

The *VerticalMap* applies the mapping function *f* to each edge of each path of the input set, where, the applications are independent among the paths. Therefore, for any input set $A =< a, \preccurlyeq_A >$, the aggregation, transformation, and combination mapping functions are defined as follows:

- **Transformation:**

$$VerticalMap(A, f) = \bigcup_{path \in paths(A)} \{\{< m_i, m_j > | < m_i, m_j > = f(< item_i, item_j >) \wedge < item_i, item_j > \in path\}\}$$

- **Aggregation:**

$$VerticalMap(A, f) = \bigcup_{path \in paths(A)} \{\{< m_i, m_j > | < m_i, m_j > = f(< it_1, it_2 >, f(\dots (f(< it_{n-1}, it_n >)) \wedge < it_i, it_j > \in path \wedge 1 \leq i \leq |path| \wedge\ 1 \leq j \leq |path|\}\}$$

- **Combination:**

$$VerticalMap(A, f) = \bigcup_{path \in paths(A)} \{\{< m_i, m_j > | < m_i, m_j > = f(< it_1, it_2 >, \dots, < it_{n-1}, it_n >) \wedge < it_i, it_j > \in path \wedge 1 \leq i \leq |path| \wedge\ 1 \leq j \leq |path|\}\}$$

To the best of our knowledge, the usage of the *VerticalMap* function is often associated with the *Correlate* function. Therefore, we exemplify the application of this operation along with the correlation examples.

Correlate

Description: finds all intermediary pairs of items connecting all source items to all target items, i.e., the Cartesian product of the leaves of the source and the target sets. Each path from each source to each target item (many-to-many) is a different path in the result set tree.

Signature: *Correlate(A, B): R × R →R*

Formal Description: let *A* and *B* be two exploration sets, *rs* be the root item of the result set, and *n* stands for an arbitrary path length. Let *Prd=lf(A)×lf(B)* be the Cartesian product of the leaf items of *A* and *B*.

The set of paths connecting each pair $< item1, item2 > \in Prd$ is defined by:

$$Correlate(A, B) = \{< rs, \{< i_1, i_2 >, \dots, < i_{n-1}, i_n >\} > | \\ \forall i_j, i_{j+1} < i_j, i_{j+1} > \in R \wedge < i_1, i_n > \in Prd \wedge 1 \leq j \leq n-1\}$$

The result set of the *Correlate* operation is a set of paths from each origin to each target item where origins are the children set of the root and the targets are the leaves.

Examples:

Let *T = {<st,{p1, p2, p3, p4}>}* be set of scientific publications, *A = {<sa,{a1, a2}>}*, and *F = {<sf, {f1, f2, f3}>}* be a set of authors' affiliations. Let *:Author={<p1,a1>,<p2,a1>,<p3,a2>,<p2,a2>,<p3,a3>,<p4,a3>}* be a relation between publications and authors. Let *:Affiliation={<a1,f1>,<a2,f1>,<a3,f2>}* be a relation between authors and affiliations.

Example 1: find connections between the publication *p1* and the affiliation *f1*:

{<rs,<p1, <a1, f1>>>} ← Correlate({p1}, {f1}), where d1 stands for the path domain and {p1, a1, f1} is the path connecting the two items.

Example 2: find connections between the publication *p2* and affiliation *f1*:

{<rs, <p2, {<a1, f1>, <a2, f1>}>>} ← Correlate({p2}, {f1}), where d1 and d2 are the domains for the two paths connecting p2 and f1.

We can notice that there are two paths from *p2* to *f1*, one that passes through *a1 (p2→ a1→ f1)* and another that passes through *a2* (*p2→ a2→ f1)*.

Example 3: find many-to-many connections between the sets *{p2, p3}* and *{f1, f2}*:

{<rs, {<p2, {<a1, f1>,<a2, f1>}>, <p3, {<a2, f1>, <a3, f2>}>}>}← Correlate({p2, p3}, {f1, f2}), where the correlation is carried out for {p2, p3} ×{f1, f2}.

Here we simplify the definition of the correlation operation in order to avoid excessive details on the operations model and keep it abstract. However, the correlation operation can be specialized with at least two additional patterns. The first parameter is the maximum distance between the origin and the target (ARAUJO *et al.*, 2010) (HEIM; LOHMANN; STEGEMANN, 2010). The second parameter is a path pattern, which is matched with each path. The operation returns only the paths that match the pattern. A language for pattern definition can be found in (PRZYJACIEL-ZABLOCKI *et al.*, 2011). A specialized signature for the *Correlate* operation is as follows:

Correlate(A, B, Pattern, maxLengh)

This chapter describes each exploration action as an atomic operator. However, we observed that some compositions of operators are very common among tools. For example, compositions of *Pivot* and *Refine* are implemented in the majority of faceted search tools. This composition is used to integrate navigation and filtering actions. Another example is *Pivot* and *Rank*, where, when the user navigates to a set of items, they are usually presented as an ordered set.

Interesting combinations also occur with the *Correlate* operation, since the discovered paths can be refined, ranked, and transformed. When combined with the *Refine* operation, the user can apply filters that leverage the analysis of connection patterns between the exploration items. For example, suppose a set of pairs that connects two hypothetical politicians *pol1* and *pol2*. The user could apply a filtering pattern to keep only paths that contain at least one intermediary node of type "Company". Let *a(%item)* be the set of all ancestors. This task can be expressed as the following composition:

```
Refine(
        Correlate({pol1}, {pol2}),
        contains(:Type[a(%item)], :Company)
)
```

The composition above filters all paths having at least one ancestor of the tail items typed as *:Company*. It is also possible to filter the paths using more complex path patterns. For example, filtering paths that comprise donations of Companies can be expressed by the following path pattern in the *Refine* operation:

Refine(

Correlate({pol1}, {pol2}),

<equals(pol1), equals(:Type, :Person), equals(:Type, :Company)

,equals(:Type, :Donation), equals(pol2)>

)

The composition above filters paths where the politician *pol1* is eventually associated to a person who owns a company that makes donations to the politician *pol2*.

Another possibility for *Correlate* compositions is to apply *Maps* to its results. For example, the user may want to map each path to a higher level of abstraction, where the relations $<item_i, item_j>$ of each path from the origins to the targets is mapped to a relation between their types. Let *:Type* be a hypothetical relation between items and their types. The following composition expresses this task:

VerticalMap(

Correlate({pol1},{pol2}),

f(%<item1,item2>)=<:Type[item1],:Type[item2]>

)

If we consider the specialized case of correlation that accepts a path pattern – *Correlate(A, B, Pattern, maxLenght)* – The composition of *VerticalMap* with *Correlate* can be used to generate the abstract path patterns that can be the input of another correlation operation or a query over the dataset. In fact, this composition describes the *Fusion* tool (ARAUJO *et al.*, 2010), where paths, such as the one presented in Figure 14, can be transformed into the abstract pattern of Figure 15 for future correlations.

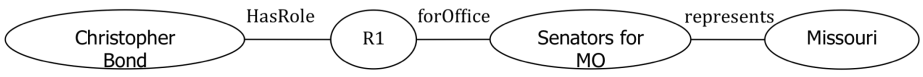

Figure 14 - Path example that correlates the Senator *Christopher Bond* with the state of *Missouri* in Gov.Track.Us[8] dataset (ARAUJO *et al.*, 2010)

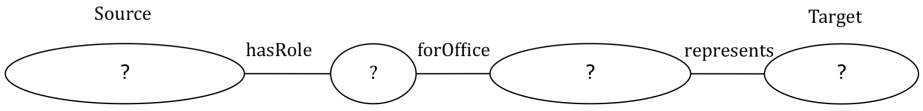

Figure 15 - Abstract path pattern that can be generated by a *VerticalMap* (ARAUJO *et al.*, 2010)

[8] https://www.govtrack.us/

Moreover, other *Correlate* compositions can be considered to improve the expressivity of *Fusion*, such as the application of refinements over a set of abstract paths or intersections and differences to compare the similarities and differences between paths. One of the key contributions of the framework is to leverage such discussions under a common understanding of the operators.

4.5.Reusing Explorations

The reuse of explorations is achieved by reevaluating functional compositions against a different set of arguments for the operations' parameters. Since the functional compositions are intentional descriptions of the exploration, the reevaluation of a functional composition is an intention-oriented operation.

The exploration process is represented as a graph, where nodes are exploration states and edges are state transitions caused by the execution of an exploration operation. In order to reuse a composition, the user can select the whole graph or a sub-graph, define new values for some operation parameters and generate a new set of states by reevaluating the operations against the new arguments. Henceforth, we define the intentional description of exploration states, exploration graphs, and the *Eval* function for reuse operations.

Definition 15: Intentional Description

The evaluation of each exploration operation generates a new exploration state, which is defined by the operation name and the parameter attributions. The invocation of an exploration operation is a n-tuple of arguments for its evaluation in the form:

$$<OperationId, Arg_1, ..., Arg_n>$$

The first element is the operation identifier, and the remaining arguments are values for the parameters in the same order of their definitions. Therefore, Arg_1 is the value for the first parameter, which is always the input set, and Arg_n is the value for the nth parameter. This is equivalent to a set of ordered pairs of parameters and arguments:

$$\{<Operation, OperationId>, <Param_1, Arg_1>, ...<Param_n, Arg_n>\}.$$

As an example, consider a initial state S_0 whose evaluation generates an exploration set of publications $P = \{<sp, \{p1, p2, p3\}>\}$ and the relations *:Author*

and *:Affiliation* that respectively relates publications with authors and authors with their affiliations. The state generated when the user pivots from S_0 to the set of authors' affiliations is expressed as follows:

$$<Pivot, S_0, :Author:Affiliation> \equiv S_0.Pivot(:Author:Affiliation)$$

In the pivot invocation above, we define the input exploration set by the id of the state whose evaluation generates it. Therefore, instead of using *P*, we use S_0, which evaluates to the set *P*. The intentional description of the state is the tuple *<:Pivot, S_0, :Author:Affiliation>*, where *:Pivot* is the identifier of the operation, S_0 is the value for the input state parameter, and *:Author:Affiliation* is the argument for the *Relation* parameter of the *Pivot* operation. The remainder of this section analyzes the operations exclusively from the point of view of the intentional descriptions. Therefore, we abstract the exploration trees generated by the execution of the operations and focus on their intentions and state dependency relations. We call these intentional descriptions *Bindings*. For the next definitions, let *B* be the set of all possible bindings for exploration.

Definition 16: Exploration Graph

An exploration graph $G \subseteq Dep$ is any sub-graph of the state dependency relation *Dep* that composes an exploration process (Section 1.2). Consider the following script:

1. $S_1 \leftarrow S_0.Pivot(:Author)$ *//pivot from publications to the set of authors*
2. $S_2 \leftarrow S_1.Refine(equals(:Affiliation, "PUC\text{-}Rio"))$ *//filter authors affiliated to PUC-Rio*
3. $S_3 \leftarrow S_2.Group(:ResearchArea)$ *// Group PUC-Rio authors by research area*

In the exploration above, S_1, S_2, and S_3 stand for exploration states in the form *<OperationId, Arg_1,..., Arg_n>*. The exploration process *E*=<St, Dep> is represented as follows:

$$E=<\{S_0, S_1, S_2, S_3\}, \{<S_0, S_1>, <S_1, S_2>, <S_2, S_3>\}>$$

The compositions of the exploration process *E* are $\{<S_0, S_2>\}$, $\{<S_1, S_2>\}$ and $\{<S_2, S_3>\}$. Next we define the *Eval* operation for reevaluating exploration graphs.

Eval

Description: the *Eval* operation receives a composition $G \subseteq Dep$ and a set of bindings for the states and generates a new set of exploration states that are reevaluations of the operations against the set of bindings.

Signature: *Eval(G, SB): Dep*^{n} × *(St×B)*^{n} → *St*^{n}, such that, *G* is an exploration sub-graph and *SB is a set of pairs* in the form *<State,Bindings>* that represents states in the first position and the new bindings for their reevaluations in the second position.

Formal Description: we define this operation as the following algorithm. Let *heads(C)* be the starting states of the sub-graph, i.e., the states whose input state is outside the sub-graph being reevaluated. Let *tails(C)* be the ending states of the sub-graph, i.e., states that are not input of any other state of the composition. Let *EvalState* be the function that receives a state, traverses the composition, starting from the state, and reevaluates each operation against the new bindings if they are defined. The *Eval* and the *EvalState* functions are defined as follows:

```
FUNCTION Eval(C, SB)
      resultStates ← empty array
      FOR ALL state in tails(C)
            resultStates.push(call EvalState(state, SB))
      END FOR
      RETURN resultStates
END FUNCTION

FUNCTION EvalState(state, SB)
      Inputs ← empty array
      IF state in heads(C) THEN
            Inputs ← state.inputs
      ELSE
            FOR all input in state.inputs
                  Inputs.push(call EvalState(input, SB))
            END FOR
      END
      IF there is a binding b in SB for state
            Operation ← b.operation
            RETURN call Operation(Inputs, b)
      ELSE
            RETURN call Operation(Inputs, state.bindings)
```

```
      END IF
END FUNCTION
```

In the algorithm above, *state.inputs* refer to the input states of a state, *state.bindings* denotes the bindings of the state, and *b.operation* is a reference to the exploration operation described in the state bindings *b*. The tail states are the starting points of the reevaluation. Each tail state is passed to the *EvalState* function, which recursively traverses the graph and executes the operations. Each execution returns a new state that is used as input for the next state. The process stops when all states in the sub-graph are reevaluated. Moreover, before reevaluating a state, the algorithm verifies if there are new binding definitions for this state. If so, the operation is executed for the new bindings *b* – *call Operation(Inputs, b)*. Otherwise, the operation is execute with the same bindings but, having the new states as inputs – *call Operation(Inputs, state.bindings)*.

As an example, consider the following composition that compares the research areas in common between researchers affiliated to PUC-Rio and UFRJ. Let the state S_1 = <*Refine, equals, :Type, "Author"*> be a initial state, which generates a set of authors $\{a_1, ..., a_n\}$ by the evaluation of a *Refine* operation. The steps are as follows:

1. $S_2 \leftarrow S_1$.Refine(equal(:Affiliation, "PUC-Rio"))
2. $S_3 \leftarrow S_1$.Refine(equal(:Affiliation, "UFRJ"))
3. $S_4 \leftarrow S_2$.Pivot(:ResearchArea)
4. $S_5 \leftarrow S_3$.Pivot(:ResearchArea)
5. $S_6 \leftarrow S_4$.Intersect(S_5)

In order to reuse the composition to obtain a comparison of research areas for researches from PUC-RS and UFMG, the reevaluation function is as follows:

NewStates ← *Eval({<S_2, S_3>, <S_2, S_4>, <S_3, S_5>, <S_5, S_6>}, {<S_2, Refine, equals, :Affiliation, "PUC-RS">, <S_3, Refine, equals, :Affiliation, "UFMG">})*

In order to simplify the representation of the *Eval* operation, we use a simplified notation expressing only the argument replacements defined by the user, represented by the replacement operator "$". The reevaluation above is also represented as:

NewStates ← *{$S_{2..6}$}.Eval(S_2."PUC-Rio"$"PUC-RS", S_3."UFRJ"$"UFMG")*

Figure 16 shows a visual representation of the states and dependencies and the order of state reevaluations.

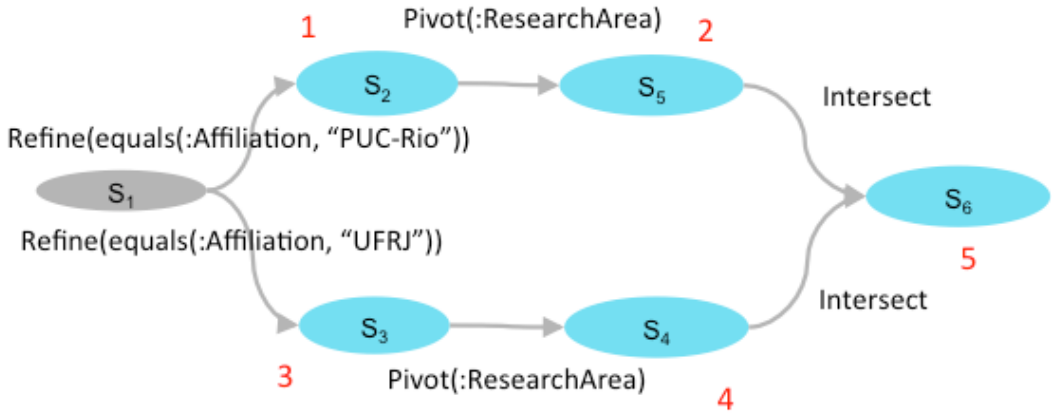

Figure 16 - Exploration graph for finding research areas in common between researchers affiliated to PUC-Rio and UFRJ with reevaluation ordering.

The highlighted nodes form the sub-graph that will be reevaluated. The head states of the sub-graph are the states S_2 and S_3 since their inputs are outside the reevaluation scope: *heads(C)* = {S_2, S_3}. The tail state is the state S_6 since it is not the input of any state in the sub-graph being reevaluated: *tails(C)*={S_6}. Therefore, the *Eval* operation traverses the sub-graph starting from S_6. When a head node is reached, it is reevaluated for the new bindings (if defined) and the result state will be the input for the next reevaluation recursively. The intersection executed in S_6 is the last state to be reevaluated, since it depends of the reevaluation of all the other nodes in the sub-graph.