Path Pattern Union

Discussion Paper: Individual Experts’ Contribution, WG3, Arusha, September 2019

Neo4j Query Languages Standards & Research Team

Continuing on [JCJ-029] and responding to [sql-pg-2019-0033]
To begin...

- Many months of discussions...and papers
  - Here are some of the papers
    - ytz036
    - bne034
    - jcj015
    - jcj019
    - jcj029
    - ..others in the SQL Ad Hoc
- Presented over multiple SQL Ad Hoc meetings
  - Overall broad concurrence within the group
Outline

- Introduction
- Overview/recap of core concepts of the pattern language
- Path pattern union
- Examples
- Path macros
Introduction

Path pattern union (PPU): an operator to express choice between alternative path patterns using (“|”) called.

pattern1 | pattern2 | ...

We propose that “|” replaces the previously proposed operators Multiset Alternation and Set Disjunction.

- We show how to achieve M.A. later on

We focus on the semantics of PPU:

- Collective understanding within the group
- Provide a base for further discussion.
- It is not yet in the form needed by standard:
  - When syntax productions are described, the production rules have been simplified. This is intended for ease of exposition.

And… the name?

- “Path pattern union” is a name of convenience. We chose it because it describes expressing a union between path patterns
- Suggestions for better names would be welcome
Overview of the pattern language:
Recapping core concepts & building blocks
Vertex Patterns

A Vertex pattern expresses:

- a variable used to identify the pattern and to reference vertices matched by the pattern,
- a label expression describing the labels a matching vertex must have, and
- a set of general predicates that a matching vertex must satisfy
  (typically these are predicates on the properties of the vertex).

While the label expressions and predicates are syntactically distinguished from one another, semantically they form a single set of predicates that a matching vertex must satisfy. From a semantic perspective we thus think of the Vertex pattern as being constituted by a vertex variable, with predicates expressed over that variable.

Vertex patterns and Edge patterns form the set of Element patterns that are used for matching graph elements.

A Vertex pattern is a Path pattern of length zero (0), with matches consisting only of a single vertex.

Example:

(bob IS Person WHERE bob.name = 'Bob')
Edge patterns

*Edge patterns*, together with *Vertex patterns*, constitute the *Element patterns* for matching the elements of the graph and from which other patterns are formed.

An *Edge pattern* expresses:
- a variable used to identify the pattern and to reference edges matched by the pattern,
- a label expression describing the labels that a matching edge must have,
- a set of general predicates that a matching edge must satisfy, and
- the *Vertex patterns* that vertices connected by the edge pattern must satisfy.

*Edge patterns*, by connecting two *Vertex patterns*, specify joins between matching vertices and the matching edge in such a way that:
- the vertices matching the source *Vertex pattern* of the *Edge pattern* are joined with the source vertex of edges matching the *Edge pattern* on the identity of the vertex, and
- the vertices matching the target *Vertex pattern* of the *Edge pattern* are joined with the target vertex of edges matching the *Edge pattern* on the identity of the vertex.

Example:

(bob)-[e:loves WHERE e.since = '1984']-(sue)
Edge patterns (continued…)

An *Edge pattern* is a *Path pattern* of length one (1), with matches consisting of a single edge in direction from left to right in the syntactic expression of the pattern..

Within a *Path pattern* it is allowed to abbreviate an edge pattern that does not specify any explicit constraints about its source and target vertices by omitting the source vertex pattern and the target edge pattern:

\[
\text{()}-[\text{e:loves WHERE } e.\text{since} = 'forever']-()\]
Joining paths by concatenation

Two Path patterns may be combined to form a Path pattern by sharing one Vertex pattern:

\[(a) - [e] -> (b) - [f] -> (c)\]

This combines the Path patterns formed by the Edge patterns for ‘e’ and ‘f’ by sharing the Vertex pattern for ‘b’.
Joining paths by juxtaposition

Two Path patterns can be combined to form a Path pattern by stating them juxtaposed next to one another.

The combination of these patterns are formed by joining the adjacent Vertex patterns of the juxtaposed Path patterns to one another:

\[(a) - [e] \rightarrow (b) \quad (c) - [f] \rightarrow (d)\]

The Path patterns formed by the Edge patterns for ‘e’ and ‘f’ are combined through the Vertex pattern for ‘b’ being joined with the Vertex pattern for ‘c’ (i.e. b and c must be matched to the same vertex in the graph).

Why do we mention them?

- Later we show how juxtaposed patterns naturally arise during the course of quantification expansion.
- We are not proposing that it must be possible to express juxtaposed patterns explicitly, but their raw syntactic expression is useful in discussing the semantics.
Path patterns

Path patterns are formed by:

- **Vertex patterns** - forming a pattern for a path of length 0.
- **Edge patterns** - forming a pattern for a path of length 1.
- **Path Patterns** joined by **Concatenation**.
- **Path Patterns** joined by **Juxtaposition**.
- **Quantified Path Patterns** - a Path Pattern repeated a number of times in a given range.
- **A Union of Path Patterns** - expressing that a matching path must match one of several alternatives (this proposal).
- Expressions limiting which qualifying paths to keep. This ranges from a simple **WHERE** to specify a predicate for the path*, to the ability to specify that only the shortest paths or the cheapest paths based on a cost function should be accepted.

*The ability to have **WHERE** associated with a Path pattern has real utility, since the Path pattern is a context in which elements can be kept as singleton variables, whereas in the **WHERE** of MATCH, variables under quantification will be turned into group variables. This is in contrast to **WHERE** in Vertex patterns and Edge patterns, which are purely a syntactic convenience, as these could equally well be expressed in **WHERE** associated with a Path pattern.
Path patterns (continued…)

A Path pattern may also express the binding of a path variable, as well as predicates over all the variables bound by a Path pattern.

Owing to the compositional nature of Path patterns, one path variable could capture a subpath of another Path pattern.

This example shows the use of WHERE with a Path pattern to express conditions spanning multiple elements:

```cypher
// a path from x to y through progressively older friends
(x) -[:KNOWS]-(a) -[:KNOWS]-(b) WHERE a.age < b.age * (y)
```
Cartesian products

A graph pattern can be composed from multiple Path patterns separated by a comma (“, ”).

The result is the Cartesian product between the paths matching each of the Path patterns.

Typically these Cartesian products are constrained through Natural joins to form conjunctive patterns.

A conjunctive pattern:

\[(a) \rightarrow [e] \rightarrow (b),\ (b) \rightarrow [f] \rightarrow (g)\]

This would form a full Cartesian product:

\[(a) \rightarrow (b),\ (c) \rightarrow (d)\]
Natural joins

If the same variable is used in multiple parts of a pattern, this stipulates a *Natural join* between those parts on that variable. The content of that variable has to be the same between the two paths.

Matching multiple edges from the same vertex:

\[
(v) - [\text{IS left_of}] \rightarrow (r), \quad (v) - [\text{IS right_of}] \rightarrow (l), \\
(v) - [\text{IS above}] \rightarrow (b), \quad (v) - [\text{IS below}] \rightarrow (a)
\]

Intuitively, using the same variable in more than one place means that the matching data in all occurrences of that variable *has to be the same* for a valid match.

Since predicates for a pattern may be expressed for any occurrence of the same variable, the predicates for that variable are combined conjunctively, except if the two occurrences are separated by a *path pattern union* ("|"), in which case the predicates are effectively combined disjunctively (we show this later).

Natural joins can be expressed within a single *Path pattern* through the repetition of a variable within the *Path pattern*. This means that a matching path has to revisit the same element, i.e. match that element in more than one place of the path. The simplest example of this is a pattern that matches a loop:

\[
(v) - [\text{loop}] \rightarrow (v)
\]
Natural joins

More complex examples are of course possible, such as describing a cycle of alternating RED and BLUE edges:

\[(v) \rightarrow \neg[:RED] \rightarrow \neg[:BLUE] \rightarrow \neg(v)\]

It is also possible to repeat other kinds of variables, such as Edge pattern variables:

\[(a)-[e:Knows]\rightarrow(b), (x)<-[e]-(y)\]

This has the side effect of joining the ‘x’ vertices with the ‘b’ vertices and joining the ‘y’ vertices with the ‘a’ vertices. Since the ‘e’ edge has to be the same in both places it has to have the same source vertices and the same target vertices in both places, thus the ‘a’ vertices have to be the same as the ‘y’ vertices and the ‘b’ vertices have to be the same as the ‘x’ vertices.

It is well worth clarifying how natural joins relate to singleton and group variables [BNE-034]. The semantics described in this section pertain only to singleton variables. This means that:

- A singleton variable may not be joined with a group variable.
  This applies to variables that are singleton vs group within the same scope.
- A group variable may not be joined with another group variable. For example, this would be forbidden (x is the variable under consideration): \[((a) \rightarrow (x) \rightarrow (b)) \ast \rightarrow (m) \rightarrow (x) \rightarrow (n)) \ast\]

A variable \(v\) in an inner scope is a singleton variable.
In an outer scope (outside a quantified pattern), \(v\) is a group variable.
A global singleton is a variable that is not quantified anywhere within the query.
Quantification

Path Patterns may be quantified. The canonical syntactic form of a quantifier is

\{n, m\}

which means repeat the quantified path pattern between \(n\) and \(m\) times.

- If \(n\) is omitted it defaults to \(0\)
- If \(m\) is omitted it defaults to \(\infty\).

In total, the following syntactic forms of quantifications are allowed:

- \(*\) - repeat the quantified path pattern \(0\) or \(\infty\) times
- \(+\) - repeat the quantified path pattern \(1\) or \(\infty\) times
- \(?\) - repeat the quantified path pattern \(0\) or \(1\) times ("optional")
- \{\(n, m\}\} - repeat the quantified path pattern between \(n\) and \(m\) times
  - if \(n\) is omitted it defaults to \(0\)
  - if \(m\) is omitted it defaults to \(\infty\)
- \{\(n\}\} - repeat the quantified path pattern exactly \(n\) times

These quantifications adhere to the semantic equivalences we describe in the next few slides.
Semantic equivalence of \{n\} (continued...)

\{n\} is semantically equivalent to the pattern repeated \(n\) times.

Given a pattern \(\alpha\), then \(\alpha\{n\}\) is equivalent to:

- the empty pattern (\(\varepsilon\)) when \(n = 0\).
- \(\alpha \ \alpha'\{n-1\}\) when \(n > 0\).
  - \(\alpha'\) denotes the same pattern as \(\alpha\) in which fresh variables have been assigned

Unrolling the recursion fully for a few cases, we arrive at:

- \(\alpha\{0\}\) expands to \(\varepsilon\) (this should not be allowed, syntactically)
- \(\alpha\{1\}\) expands to \(\alpha\)
- \(\alpha\{2\}\) expands to \(\alpha \ \alpha'\)
- \(\alpha\{3\}\) expands to \(\alpha \ \alpha' \ \alpha''\)
- ...and so on...
Quantification: Semantic equivalence of \{n\}

\(\alpha'\) (and \(\alpha''\), etc.) indicates separate naming contexts, or namespaces, which qualify the names of variables for each repetition of the pattern.

- The names of variables in one repetition do not interfere with the names of variables in another repetition.

The results bound to the same variables in different naming contexts are however collected together into the same grouped variable on the outside of the pattern.

The same naming context will appear in multiple operands of the implied path pattern union. However, this does not cause any interference, since the patterns in each operand are of different fixed shape (each operand is one 'hop' longer than the preceding operand) and cannot match at the same time.
Quantification: Semantic equivalence of \{n,m\}

\{n, m\} - is considered to be the path pattern union of each fixed repetition of the pattern.

Given a pattern \(\alpha\), then \(\alpha\{n, m\}\) (where \(m\) may be infinity) is the path pattern union of all patterns in the set:

\[
\{ \alpha\{k\} \mid n \leq k \leq m \}
\]

In the case where \(m\) is infinite (i.e. unspecified), this leads to an infinite set.

We are not suggesting that quantification expansion to a possibly infinite (or extremely large, albeit bounded) set of union operators is done statically at compile time.

- We are merely using this “expand to a set of unions” device in order to express the semantics.

Spelling out the path pattern union of this set (which we cannot do if \(m\) is infinite), we get:

\[
\alpha\{n\} \mid \alpha\{n+1\} \mid \ldots \mid \alpha\{m-1\} \mid \alpha\{m\}
\]
Semantic equivalence of \{n,m\} (continued...)

For a pattern \(((t) - [E] -> (h)) \{0, 3\}\) the result of this expansion would be

\[
\epsilon\\
| (t) - [E] -> (h)\\n| (t) - [E] -> (h) (t') - [E'] -> (h')\\n| (t) - [E] -> (h) (t') - [E'] -> (h') (t'') - [E''] -> (h'')
\]

Any path that matched one of these four path conditions would satisfy the operand.

Our semantics extend straightforwardly to predicates on the pattern which is quantified (we show this later).
Quantification: Semantics of the empty pattern ($\varepsilon$)

The empty pattern (denoted by $\varepsilon$) matches nothing on its own, and does not affect other patterns when juxtaposed. The empty pattern arises from zero repetitions of a quantified pattern, and behaves as if no pattern had been written at all.

This means that:

- $\text{MATCH } \varepsilon$ has no solutions and produces no results
- Given arbitrary patterns $\beta$ and $\gamma$:
  - $\varepsilon \beta \equiv \beta$
  - $\beta \varepsilon \equiv \beta$
  - $\beta \varepsilon \gamma \equiv \beta \gamma$
  - $\varepsilon | \beta \equiv \beta$

  This does not mean that $\varepsilon$ is insignificant in path pattern union, the distributivity of the surrounding patterns across the path pattern union (defined in Canonicalization rule 6 later on) needs to be applied before $\varepsilon$ can be removed.

- $\varepsilon, \beta$ has no solutions and produces no results
Semantics of the empty pattern ($\varepsilon$) (continued...)

We distinguish between two types of quantified patterns:

- **Partial pattern**: This is a pattern where the quantification is only expressed over the adjoining edges and intermediate vertices in a path; i.e. the source and target vertices (i.e. the two ultimate ‘endpoint’ vertices) are not included in the quantification.
  
  Examples include (quantification scope emboldened):
  
  $$(x:\text{Person}) \ ((a)\rightarrow(b)) \star \ (y) \text{ and } \ (a)\rightarrow\star \ (b)$$

- **Full pattern**: This is a pattern where the quantification is expressed over the entire pattern. That is, the source and target vertices are included in the quantification.
  
  An example - with the quantification scope emboldened - would be $$( (a) \rightarrow (b) ) \star$$

Evaluating $\varepsilon$ on a partial pattern would essentially mean that one is expressing a match on two juxtaposed vertex patterns (which must be matched to the same vertex in the graph). For example, $\text{MATCH } (a) \rightarrow \{0\} \ (b)$ is equivalent to $\text{MATCH } (a) (b)$. This extends to versions containing predicates:

- $\text{MATCH } (x:\text{Person}) \ ((a)\rightarrow(b)) \{0\} \ (y) \Leftrightarrow \text{MATCH } (x:\text{Person}) \ (y)$
- $\text{MATCH } (a)\rightarrow\{0\} \ (b) \text{ WHERE } a.\text{foo}=12 \Leftrightarrow \text{MATCH } (a) \ (b) \text{ WHERE } a.\text{foo}=12$

By contrast, evaluating $\varepsilon$ on a full pattern will always reduce to evaluating $\text{MATCH } \varepsilon$. One can therefore argue that, from a user’s point of view, the Kleene * operator on a full pattern makes no sense, and that the permitted lower bound ought always to be at least one; i.e the Kleene + operator.

- A full pattern can always be transformed into a partial pattern by juxtaposing a source and target vertex pattern around the quantification. Thus, the full pattern $((a) \rightarrow (b)) \star$ becomes the partial pattern $(x) \ ((a) \rightarrow (b)) \star \ (y)$
CHEAPEST & SHORTEST

Path patterns may express that only the shortest (by number of edges in the path) or cheapest (by the sum of the specified cost functions for each sub-pattern of the path pattern along the matched path) paths are to be matched by the pattern.

At this stage, we believe that these features follow on straightforwardly from our proposed semantics without the need for special treatment at this point.
Path pattern union
Basic outline of PPU semantics

1. Transform the syntactic form of the operands into a canonical form consisting of fixed patterns:
   - Each node, edge (i.e. element) and path pattern is assigned a variable.
   - If no variables are explicitly provided in the syntactic form, implicit variables are assigned so that element or path patterns in the same position in different operands of PPU are assigned the same implicit variable.

2. We now have two or more fixed pattern operands, where all variables have been assigned to all element and path patterns. Then:
   - The evaluation of each operand of a Path pattern union results in a set of bindings - of elements to the variables - corresponding to the paths matching the operand. I.e each operand produces a (binding) table of matches corresponding to the pattern.
     - Each variable in the operand corresponds to a column in the table.
     - Each row corresponds to a matching solution; i.e. variable bindings induced by the solution
   - Undertake a regular relational set union between the tables
     - Each of these tables is extended to have columns matching the complete set of variables from any operand (“outer union”). I.e. If a variable is missing in an operand, then any solution row induced by the operand will have NULL assigned to the variable.
     - The semantics of “outer union” can be implemented without having to materialize these tables and deduplicate the rows. Given complete sets of canonical (fixed-length) operands for all operands, it is straightforward to recognize when two operands produce the same bindings for the same positions and combine the predicates of these into a single operand.
Basic outline of PPU semantics (continued...)

Thus, each set of results - induced by each operand - are combined by the path pattern union into a final set of results.

The intuition behind this is that if the exact same set of variables - say, a, b and c - was used in more than one operand - say, op₁ and op₂ - and the same path p in the data was matched by both op₁ and op₂, then the binding of the elements in p to a, b and c will appear once in the final result set.

Operating under set semantics would mean that the path pattern union operation would be fully composable.

In summary, a single row is produced for every unique “matched path and variable-binding” combination.
Canonicalization of patterns

Patterns are canonicalized according to the following rules (iteratively until a fixed point is reached):

1. **Vertex patterns** are always canonical.

2. Short form **Edge patterns** are canonicalized to include a (empty) body. 
   I.e. `(,) -> ()` is canonicalized to `(,) - [ ] -> ()`

3. **Path patterns** consisting of an **Edge pattern** without vertices are canonicalized by inserting empty vertices on each side of the **Edge pattern**.

4. **Quantified patterns** are canonicalized by ensuring that the quantified pattern is enclosed in parentheses, and by canonicalizing the quantifier to the `{n,m}`- and `{n}`-forms according to the semantic equivalences described earlier. This ensures that all quantifications are converted into (potentially infinite) **Path pattern unions** of **fixed patterns**.
5. In Juxtapositioned patterns, remove any $\varepsilon$.
Remaining juxtapositioned vertex patterns - where at least one of which lacks a vertex variable declaration - are consolidated into a single vertex pattern, retaining any vertex variable declaration and WHERE clause.

If both vertex patterns have label expressions, they are combined using conjunction ($\&$).

6. Apply distributivity of Path patterns across Path pattern union (i.e. that $\alpha (\beta \mid \gamma) \delta$ is equivalent to $\alpha\beta\delta \mid \alpha\gamma\delta$) to “lift” the Path pattern union to the top level of the pattern.

7. Combine operands of Path pattern union that have the same variables in the same places.
(This rule applies after implicit variables have been assigned.)
Predicates (label expressions and WHERE) from the combined operands are combined by logical OR.

Note that the predicates of each operand must be grouped together and combined as a whole with the predicates from the other operands (this becomes clearer when considering all predicates (such as label expressions) in their canonical form - expressed in a "WHERE" clause.).
Variable assignment

The semantics of pattern matching is fundamentally based on the variables in the pattern.

Reusing the same variables in multiple parts of the pattern is how joins between patterns are expressed.

For *path pattern union*, using the same variables between operands expresses that the patterns are to be union compatible (if they match the same path).

With the semantics of pattern matching based on the variables in the pattern, the semantics require a complete assignment of variables. All vertex patterns, edge patterns, and path patterns must have variables assigned for the semantic expression of what matches the pattern.

Since we want to syntactically allow the user to leave out variables that are not of importance for what the query asks for, we introduce a notion of *implicitly assigned variables* (or, shorter, *implicit variables*) as well as a set of rules for how the implicit variables are assigned.
Variable assignment (continued...)
Variable assignment (continued...)

Consider two canonicalized path patterns, \( pp_1 \) and \( pp_2 \), each of which is an operand to the path pattern union operator in a pattern. An element pattern \( e_1 \) in \( pp_1 \) is defined to be positionally compatible with an element pattern \( e_2 \) in \( pp_2 \) if:

- Proceeding from the left of the pattern, and ignoring the directionality of the pattern (induced by the edge), \( e_1 \) occurs in precisely the same position in \( pp_1 \) as \( e_2 \) in \( pp_2 \)
  - This means that if \( e_1 \) is a vertex (resp. edge) pattern, then so is \( e_2 \)

We note that it doesn't matter if the operands contain differing numbers of element patterns.

E.g. \((-[[]]->())\) and \((()->())\) both have three elements (once canonicalized), whereas \((-[[]]->()-[]-())\) has five elements.

Since these patterns are structurally incompatible, and would be assigned a different number of implicit variables, it is safe to assign implicit variables purely based on positional compatibility.

For example, the positionally compatible element patterns in the following path patterns are indicated by colour and number:

- \((e_1) - [e_2] - (e_3)\)
- \((e_1) <- [e_2] - (e_3)\)
- \((e_1) - [e_2] - (e_3) - [e_4] - (e_5)\)
- \((e_1)(e_1) - [e_2] - (e_3)(e_3)\)

Two fixed patterns are structurally compatible if they contain the same number of element patterns (where juxtapositioned vertex patterns are considered to be the same vertex pattern).
Variable assignment (continued...)  

Here are the following general principles for implicit variable assignment:

1. Before implicit variables are assigned, the pattern is canonicalized to expose all parts of the pattern that should be named.

2. Implicit variables must not interfere with any explicit variables anywhere in the query.

3. When assigning implicit variables to the operands of a path pattern union, elements at the same position of structurally compatible operands must be assigned the same implicit variables (if no explicit variable has already been assigned). This is realized by assigning the same implicit variable to elements of patterns that are positionally compatible.
   ○ For path variables, the same process applies for assigning variables positionally. However more work is needed on the algorithm for deciding where to introduce path variables.

4. Path macros are expanded with all their variables replaced with variables allocated according to the implicit variable assignment rules. This ensures that macro expansion is hygienic (i.e. that the variables of the macro do not interfere with the variables at the place where the macro is used) and provides the ability to define the semantics in such a way that the order between operands of path pattern union is independent of the contents of the operands.
Examples: fixed patterns
Eg. 1: Positionally compatible elements with identical explicit variables

\[(a \text{ IS Animal})-[e]->(b) \mid (a \text{ IS Cat})-[e]->(b)\]

- The left operand gives rise to a set containing both rows
- The right operand gives rise to a set containing the second row only.
- PPU combines both of these sets into the final set

<table>
<thead>
<tr>
<th>Operand</th>
<th>a</th>
<th>e</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left</td>
<td>1</td>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>Left, Right</td>
<td>2</td>
<td>22</td>
<td>4</td>
</tr>
</tbody>
</table>

Thus, such a pattern could be rewritten to a label disjunction, as these two forms of patterns are equivalent:

\[(a \text{ IS Animal})-[e]->(b) \mid (a \text{ IS Cat})-[e]->(b) \Leftrightarrow (a \text{ IS Animal}|\text{Cat})-[e]->(b)\]

However, caution needs to be exercised in generalizing this, and thus bears further investigation.

- Any such rewritings of a *path pattern union* \(p1\) to a simpler pattern \(p2\) using label disjunctions must not lead to solutions (for \(p2\)) which would not be valid for \(p1\).
- \((\text{IS A})-[e]->(\text{IS B}) \mid (\text{IS C})-[e]->(\text{IS D}) \Leftrightarrow (\text{IS A|C})-[e]->(\text{IS B|D}) \text{ is illegal!}\)
Eg. 2: Positionally compatible elements with different explicit variables

(a IS Animal)-[e]->(b) | (d IS Cat)-[e]->(b)

- The “outer union” semantics surface much more clearly
- For the first operand, we see the same two rows as for Example 1, but now with the addition of a third row which is induced by the second operand.
- In effect, this means we have two rows for the same path \( p \) from vertex 2 to vertex 4 via edge 22 (essentially, multiset semantics insofar as the path \( p \) is concerned, although not where the variables are concerned).
  - We recall that this is in line with our principle whereby a single row is produced for every unique “matched path and variable-binding” combination.

<table>
<thead>
<tr>
<th>Operand</th>
<th>a</th>
<th>e</th>
<th>b</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left</td>
<td>1</td>
<td>11</td>
<td>3</td>
<td>null</td>
</tr>
<tr>
<td>Left</td>
<td>2</td>
<td>22</td>
<td>4</td>
<td>null</td>
</tr>
<tr>
<td>Right</td>
<td>null</td>
<td>22</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>
Eg. 3: Positionally compatible elements, where one is implicit, and the other explicit

In the case where a pattern in one operand has an explicit variable, but the positionally compatible pattern in the other operand does not, a differently-named variable is assigned implicitly.

The pattern

(IS Animal)-[e]->(b) | (a IS Cat)-[e]->(b)  //original pattern

would be rewritten to

(_1 IS Animal)-[e]->(b) | (a IS Cat)-[e]->(b)  //rewritten pattern

<table>
<thead>
<tr>
<th>Operand</th>
<th>_1</th>
<th>a</th>
<th>e</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left</td>
<td>1</td>
<td>null</td>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>Left</td>
<td>2</td>
<td>null</td>
<td>22</td>
<td>4</td>
</tr>
<tr>
<td>Right</td>
<td>null</td>
<td>2</td>
<td>22</td>
<td>4</td>
</tr>
</tbody>
</table>
Positionally compatible elements, where one is implicit, and the other explicit (cont...)

Why can the rewriting not simply assign an implicit variable with the same name as the explicit variable to produce:

\[(a \text{ IS Animal}) - [e] -\rightarrow (b) \mid (a \text{ IS Cat}) - [e] -\rightarrow (b)\]

This would results in the same results as Example 1:
- Seems as if it may make more sense - perhaps the user simply forgot to provide an explicit variable for the first operand (and that in all likelihood this would have been intended to be the same variable)?

Attempting to unite an explicit variable with an implicit one could actually lead to problems.

Assume the pattern (and that no vertex is ever both a Dog and a Cat):

\[(\text{IS Dog}) - [e] -\rightarrow (b) \mid (a \text{ IS Cat}) - [e] -\rightarrow (b)\]

The user may very well wish to express that any vertex bound to \(a\) must always be a Cat, and would expect the results to reflect this. Of course, the user expects to match on Dog vertices as well, but would in all likelihood not expect these to be bound to \(a\), and if the results were to do this, it could lead to a lot of confusion (as to why a pattern that explicitly mandated binding to Cat vertices now also included bindings to Dog vertices).

We also think it would be odd to conflate implicit and explicit variables in this way, particularly when it comes to projection. The user left the Dog vertices unnamed - for whatever reason - and to then bind them to the same variable as the explicitly-assigned one would cause more problems than it solves.
Eg. 4: Positionally compatible elements with identical implicit variables

If two operands have no explicit variables in positionally compatible element patterns, we propose that the same implicit variable is assigned.

\[(\text{IS Animal})-[e]->(b) \mid (\text{IS Cat})-[e]->(b)\]

would be rewritten to

\[(\_1 \text{ IS Animal})-[e]->(b) \mid (\_1 \text{ IS Cat})-[e]->(b)\]

<table>
<thead>
<tr>
<th>Operand</th>
<th>_1</th>
<th>e</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left</td>
<td>1</td>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>Left, Right</td>
<td>2</td>
<td>22</td>
<td>4</td>
</tr>
</tbody>
</table>
Eg. 5: Achieving Multiset Alternation

\[(p \text{ IS Person}) \rightarrow [e] \rightarrow (q \text{ IS Person}) \mid (r) \rightarrow [f \text{ IS Knows}] \rightarrow (s)\]

<table>
<thead>
<tr>
<th>Operand</th>
<th>p</th>
<th>e</th>
<th>q</th>
<th>r</th>
<th>f</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left</td>
<td>1</td>
<td>101</td>
<td>2</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>Left</td>
<td>1</td>
<td>102</td>
<td>2</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>Right</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>1</td>
<td>101</td>
<td>2</td>
</tr>
<tr>
<td>Right</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>1</td>
<td>103</td>
<td>3</td>
</tr>
</tbody>
</table>

This is precisely the same set of rows as those mentioned in [JCJ-015]; in other words, complying with the semantics of Multiset Alternation.
Achieving Multiset Alternation (continued…)

Multiset Alternation semantics can be emulated with a change of one variable:

(a) \((\text{IS Person}) - [x] \rightarrow (\text{IS Person}) \mid -[y \text{ IS Knows}] \rightarrow\)  

(b)

<table>
<thead>
<tr>
<th>Operand</th>
<th>a</th>
<th>x</th>
<th>y</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left</td>
<td>1</td>
<td>101</td>
<td>\text{null}</td>
<td>2</td>
</tr>
<tr>
<td>Left</td>
<td>1</td>
<td>102</td>
<td>\text{null}</td>
<td>2</td>
</tr>
<tr>
<td>Right</td>
<td>1</td>
<td>\text{null}</td>
<td>101</td>
<td>2</td>
</tr>
<tr>
<td>Right</td>
<td>1</td>
<td>\text{null}</td>
<td>103</td>
<td>3</td>
</tr>
</tbody>
</table>
Achieving Multiset Alternation (continued…)

Using an alternative formulation of the query, as given in [BNE-034]:

(a) \((\text{IS Person}) \rightarrow (\text{IS Person}) \mid -\text{[IS Knows]} \rightarrow\) (b)

Written in canonical form, this pattern becomes:

\((a \text{ IS Person}) \rightarrow \_2 \rightarrow (b \text{ IS Person}) \mid (a) \rightarrow \text{[IS Knows]} \rightarrow (b)\)

After implicit variable assignment, this becomes:

\((a \text{ IS Person}) \rightarrow \_2 \rightarrow (b \text{ IS Person}) \mid (a) \rightarrow \_2 \text{ IS Knows} \rightarrow (b)\)

Catering for both use cases - i.e. allowing for either multiset semantics or set semantics - using a single operator is indeed possible. All that is required is to ensure at least one unique explicit variable is assigned to a positionally compatible element pattern in the operands.

<table>
<thead>
<tr>
<th>Operand</th>
<th>a</th>
<th>_2</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left, Right</td>
<td>1</td>
<td>101</td>
<td>2</td>
</tr>
<tr>
<td>Left</td>
<td>1</td>
<td>102</td>
<td>2</td>
</tr>
<tr>
<td>Right</td>
<td>1</td>
<td>103</td>
<td>3</td>
</tr>
</tbody>
</table>

These results would be achieved without actually evaluating each operand in turn, and then doing a set union over the two result sets (from each operand) by applying Canonicalization rule 7.
Eg. 6: Considering path variables

\[(x) \ ( (a) - [e] -> (b) \ | \ (b) <- [e] - (a) ) \ (y) \]

- Explicit variables \(a\), \(e\), and \(b\) are provided.
- \(x\) and \(y\) are provided as the start and end vertices of the matched path
  - This illustrates that the order in which a pattern is stated matters.

The introduction of implicit path variables need to provide semantics that produce results consistent with the bindings for \(x\) and \(y\), and do so even if \(x\) and \(y\) had not been present in the pattern.

Here, following the semantics of our proposal, we assign the same implicit path variable \(_1\) to both operands.

\[(x) \ (_1 = (a) - [e] -> (b) \ | \ _1 = (b) <- [e] - (a) ) \ (y) \]

<table>
<thead>
<tr>
<th>(_1)</th>
<th>(x)</th>
<th>(a)</th>
<th>(e)</th>
<th>(b)</th>
<th>(y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 11, 5</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>2, 22, 5</td>
<td>2</td>
<td>2</td>
<td>22</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>5, 55, 5</td>
<td>5</td>
<td>5</td>
<td>55</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(_1)</th>
<th>(x)</th>
<th>(a)</th>
<th>(e)</th>
<th>(b)</th>
<th>(y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5, 11, 1</td>
<td>5</td>
<td>1</td>
<td>11</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>5, 22, 2</td>
<td>5</td>
<td>2</td>
<td>22</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>5, 55, 5</td>
<td>5</td>
<td>5</td>
<td>55</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>
Considering path variables (continued…)

\[(x) \ ( (a) - [e] \rightarrow (b) \mid (b) \leftarrow [e] - (a) ) \ (y)\]

- Explicit variables \( a, e, \) and \( b \) are provided.
- \( x \) and \( y \) are provided as the start and end vertices of the matched path
  - This illustrates that the order in which a pattern is stated matters.

Here, following the semantics of our proposal, we assign the same implicit path variable \(_1\) to both operands.

\[(x) \ (_1 = (a) - [e] \rightarrow (b) \mid _1 = (b) \leftarrow [e] - (a) ) \ (y)\]

<table>
<thead>
<tr>
<th>Operand</th>
<th>(_1)</th>
<th>(x)</th>
<th>(a)</th>
<th>(e)</th>
<th>(b)</th>
<th>(y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left</td>
<td>1, 11, 5</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Left</td>
<td>2, 22, 5</td>
<td>2</td>
<td>2</td>
<td>22</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Left, Right</td>
<td>5, 55, 5</td>
<td>5</td>
<td>5</td>
<td>55</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Right</td>
<td>5, 11, 1</td>
<td>5</td>
<td>1</td>
<td>11</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Right</td>
<td>5, 22, 2</td>
<td>5</td>
<td>2</td>
<td>22</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>
Considering path variables (continued…)

<table>
<thead>
<tr>
<th>_1</th>
<th>x</th>
<th>a</th>
<th>e</th>
<th>b</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 11, 5</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>2, 22, 5</td>
<td>2</td>
<td>2</td>
<td>22</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>5, 55, 5</td>
<td>5</td>
<td>5</td>
<td>55</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>_1</th>
<th>x</th>
<th>a</th>
<th>e</th>
<th>b</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>5, 11, 1</td>
<td>5</td>
<td>1</td>
<td>11</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>5, 22, 2</td>
<td>5</td>
<td>2</td>
<td>22</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>5, 55, 5</td>
<td>5</td>
<td>5</td>
<td>55</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

Projecting out the explicit variables a, e, and b would result in what seems like a multiset.

- This is not actually the case, as each operand results in a different path (bound to the implicit variable _1) with respect to the *ordering* of the elements therein
- The path (1, 11, 5) is *not the same* as the path (5, 11, 1).

This is why patterns are not rewritten to change the order from the order in which it was written
- The order in which the user expressed the pattern is *significant* to the semantics of the pattern.

Aligning with the remarks made in [sql-pg-2019-0033] regarding the criticality of the sequential nature of paths, we note that the order of the elements within the path is very important.
Considering path variables (continued…)

If the “Multiset Alternation (M.A.) Approach”* is applied, we would get

\[(x) \ _1 = (a) - [e] \rightarrow (b) \ | \ _2 = (b) \leftarrow [e] - (a) \ \} (y)\]

*I.e. unique implicit variables are assigned for every pattern across the whole PPU

<table>
<thead>
<tr>
<th></th>
<th>(a) - [e] \rightarrow (b)</th>
<th></th>
<th>(b) \leftarrow [e] - (a)</th>
</tr>
</thead>
<tbody>
<tr>
<td>_1</td>
<td>_2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1, 11, 5</td>
<td>1, 1, 11, 5, 5</td>
<td>5, 11, 1</td>
<td>5, 1</td>
</tr>
<tr>
<td>2, 22, 5</td>
<td>2, 2, 22, 5, 5</td>
<td>5, 22, 2</td>
<td>5, 2</td>
</tr>
<tr>
<td>5, 55, 5</td>
<td>5, 5, 55, 5, 5</td>
<td>5, 55, 5</td>
<td>5, 5</td>
</tr>
</tbody>
</table>
(x) ( (a) - [e] -> (b) )

(b) <- [e] - (a)) (y)

M.A. version duplicates the row corresponding to the loop.

The variables x, a, e, b and y are explicitly projected from the query.

The duplication adds no value, and argues in favour of our proposal.

(One can still achieve M.A. through explicit assignment of variables in the way that the “M.A Approach” would assign implicit variables, but our approach seems like a better one for the assignment of implicit variables.)
Eg. 7: Operands of differing lengths

\[(a) \rightarrow (b) \mid (a) \rightarrow (\_\_\_\_\_) \rightarrow (b)\]

Implicit variables will be assigned thus:

\[(a) \rightarrow \_\_\_\_ \rightarrow (b) \mid (a) \rightarrow \_\_\_\_ \rightarrow \_\_\_\_ \rightarrow \_\_\_\_ \rightarrow (b)\]

Although the same implicit variables are assigned to all positionally compatible element patterns, this obviously results in a different number of variables in each operand.

This means the operands are structurally incompatible (irrespective of whether or not they have the same implicit variables).

<table>
<thead>
<tr>
<th>Operand</th>
<th>a</th>
<th>_2</th>
<th>_3</th>
<th>_4</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left</td>
<td>v1</td>
<td>e1</td>
<td>null</td>
<td>null</td>
<td>v2</td>
</tr>
<tr>
<td>*Left</td>
<td>v1</td>
<td>e3</td>
<td>null</td>
<td>null</td>
<td>v3</td>
</tr>
<tr>
<td>Left</td>
<td>v2</td>
<td>e2</td>
<td>null</td>
<td>null</td>
<td>v3</td>
</tr>
<tr>
<td>*Right</td>
<td>v1</td>
<td>e1</td>
<td>v2</td>
<td>e2</td>
<td>v3</td>
</tr>
</tbody>
</table>

Projecting out the explicit variables (a and b) would result in a multiset for rows 2 and 4. This is correct, as these two rows represent two different paths.
Examples: quantified patterns

Focussing on canonicalization
Example A

(a) ->* (b) | (a) -> (b)

⇔ (a) [-] ->* (b) | (a) [-] -> (b)
according to CR 2

⇔ (a) ( () [-] -> () ) * (b) | (a) [-] -> (b)
according to CR 3

⇔ (a) ε | () [-] -> () |
() [-] -> () () [-] -> () | ... ) (b) |
(a) [-] -> (b)
according to CR 4

⇔ (a) ε (b) | (a) () [-] -> () (b) |
(a) () [-] -> () () [-] -> () (b) |
(a) ... (b) |
(a) [-] -> (b)
according to CR 6

⇔ (a) (b) | (a) [-] -> (b) | (a) [-] -> () [-] -> (b)
| (a) ... (b) |
(a) [-] -> (b)
according to CR 5

⇔ _0= (a) (b) | _0= (a) [-_2] -> (b) |
_0= (a) [-_2] -> (_3) [-_4] -> (b) |
_0= (a) ... (b) |
_0= (a) [-_2] -> (b)
by assignment of implicit variables

⇔ _0= (a) (b) | _0= (a) [-_2] -> (b) |
_0= (a) [-_2] -> (_3) [-_4] -> (b) |
_0= (a) ... (b)
since we can eliminate duplicate occurrences of equivalent operands, as the set semantics of path pattern union would eliminate the duplicate results.
(according to CR 7)

The final result is equivalent to (a) ->* (b) since the other branch is redundant.
Example B

\[(a) \ ( (x) -> (y) )^* \ (b) \ | \ (a) -> (b)\]

\[\Leftrightarrow\]

\[(a) (b) \ | \ (a) (x) - [ ] -> (y) (b) \ | \]

\[(a) (x) - [ ] -> (y) (x') - [ ] -> (y') (b) \ | \ (a) \ldots (b) \ | \ (a) - [ ] -> (b)\]

according to the Canonicalization rules

\[\Leftrightarrow\]

\[._0 = (a) (b) \ | \ _0 = (a) (x) - [ _2 ] -> (y) (b) \ | \]

\[._0 = (a) (x) - [ _2 ] -> (y) (x') - [ _4 ] -> (y') (b) \ | \ _0 = (a) \ldots (b) \ | \ _0 = (a) - [ _2 ] -> (b)\]

by assignment of implicit variables

Here we see that the second operand of the original path pattern union will not be eliminated, since it binds different variables from the one of equal length produced by the Kleene star.
Example C

\[ ( (x) \rightarrow (y) \text{ WHERE } x.p < y.p )^* \mid ( (x) \rightarrow (y) \text{ WHERE } x.p > y.p ) \]

\( \iff \) \( \varepsilon \mid \)

\(( (x) - [\_2] \rightarrow (y) \text{ WHERE } x.p < y.p ) \mid \)

\(( (x) - [\_2] \rightarrow (y) (x') - [\_4] \rightarrow (y') \)

\( \text{ WHERE } x.p < y.p \text{ AND } x'.p < y'.p \)

\) \mid \)

\( \ldots \mid \)

\(( (x) - [\_2] \rightarrow (y) \text{ WHERE } x.p > y.p ) \)

according to the Canonicalization rules

\( \iff \) \( _0 = \varepsilon \mid \)

\(( _0 = (x) - [\_2] \rightarrow (y) \text{ WHERE } x.p < y.p \)

\) \mid \)

\(( _0 = (x) - [\_2] \rightarrow (y) (x') - [\_4] \rightarrow (y') \)

\( \text{ WHERE } x.p < y.p \text{ AND } x'.p < y'.p \)

\) \mid \)

\( \ldots \mid \)

\(( _0 = (x) - [\_2] \rightarrow (y) \text{ WHERE } x.p > y.p \)

by assignment of implicit variables

\( \iff \) \( _0 = \varepsilon \mid \)

\(( _0 = (x) - [\_2] \rightarrow (y) \text{ WHERE } x.p < y.p \)

\( \text{ OR } x.p > y.p \) \mid \)

\(( _0 = (x) - [\_2] \rightarrow (y) (x') - [\_4] \rightarrow (y') \)

\( \text{ WHERE } x.p < y.p \text{ AND } x'.p < y'.p \)

\) \mid \)

\( \ldots \)

since the two patterns of equal length are compatible, we can combine the predicates.

(according to Canonicalization rule 7)
Example D

\[(x) -\rightarrow (y) \star | (n) -\rightarrow (m)\]

\[\Leftrightarrow \quad _0=\varepsilon \quad | \]
\[\quad _0=(x)-[\_1]-\rightarrow (y) \quad | \]
\[\quad _0=(x)-[\_1]-\rightarrow (y)(x')-[\_2]-\rightarrow (y') \quad | \]
\[\quad \ldots \quad | \]
\[\quad _0=(n)-[\_1]-\rightarrow (m) \]

according to the Canonicalization rules and assignment of implicit variables

At this point we see that the second operand of the original path pattern union will not be eliminated, since it binds different variables from the other pattern of equal length.
Example E

\[(a) - [e] - (b) \mid (\ (a) - [e] - (b) \ ( (c) - [f] - (d)) ? ) \]

\[\Leftrightarrow\] \[(a) - [e] - (b) \mid (\ (a) - [e] - (b) \ (\varepsilon \mid (c) - [f] - (d)) ) \]
according to CR 4

\[\Leftrightarrow\] \[(a) - [e] - (b) \mid (a) - [e] - (b) \varepsilon \mid (a) - [e] - (b) (c) - [f] - (d) \]
according to CR 6

\[\Leftrightarrow\] \[(a) - [e] - (b) \mid (a) - [e] - (b) \mid (a) - [e] - (b) (c) - [f] - (d) \]
according to CR 5

\[\Leftrightarrow\] \[_0= (a) - [e] - (b) \mid _0= (a) - [e] - (b) (c) - [f] - (d) \]
by assignment of implicit variables and elimination of equivalent operands
Example F

\[(a) - [:T] ->* (:Person) \mid (a) - [:T] -> () - [:T] ->* (:Person)\]

⇔

\[(a) () - [:T] -> (())* (:Person) \mid (a) - [:T] -> (()) (()) - [:T] -> (())* (:Person)\]

according to CR 3

⇔

\[(a) \epsilon (:Person) \mid (a) () - [:T] -> () (:)Person) \mid (a) () - [:T] -> () - [:T] -> () (:)Person) \mid (a) ... (:Person) \mid (a) - [:T] -> () \epsilon (:Person) \mid (a) - [:T] -> () - [:T] -> () (:)Person) \mid (a) - [:T] -> () - [:T] -> () - [:T] -> () (:)Person) \mid (a) - [:T] -> () ... (:Person)\]

according to CR 5

This is an example of a partial pattern
Example F

(a)-[:T]->*(:Person) | (a)-[:T]->()[:T]->*(:Person)

... 

⇔

(a:Person) |
(a)-[2:T]->(_3:Person) | // A
(a)-[2:T]->(_3)-[4:T]->(_5:Person) | // B
(a)...(:Person) // C
| (a)-[2:T]->(_3:Person) | // A
(a)-[2:T]->(_3)-[4:T]->(_5) | // B
(a)-[2:T]->(_3)-[4:T]->(_5)-[6:T]->(_7:Person) |
(a)-[2:T]->(_3)...(:Person) // C

by assignment of implicit variables and elimination of equivalent operands

We note that the same pattern appears in both operands (highlighted in bold and named “A”, “B”, and “C” in comments), and that this will become equivalent to just (a)-[:T]->*(:Person), which is in line with the intuition for the initial expression of the query based on regular expression rewritings.
Path macros
Path macros

The semantics of pattern matching naturally extends to Path macros as well.

While Path macros is a topic worthy of a whole paper of its own, it is valuable to show how the implicit variable assignment interacts with Path macros and provides important and desirable traits to Path macro expansion and its interaction with path pattern union.

Path macros are hygienic.

- This means that the variables in the macro definition must not influence the variables in the pattern where the macro is expanded.

This is realized semantically by assigning new variables to all parts of the expanded macro, rewriting the references to the variables in the macro definition to use the newly assigned variables.

This is realized by the implicit variable assignment algorithm.

This is safe since the variables of the macro are not accessible outside the macro definition.

However, we might want to introduce ways of binding results computed by a Path macro to variables explicitly exposed by the macro, but that is a topic for another paper.
Basic example

Assume we have the macro `INHERITS`:

**PATH** \((x)-/INHERITS/-\)\(\rightarrow\)(\(y\)) =
  \((x)-[: A \mid B \mid C \mid D]\rightarrow(\(y\))

Now assume we have the pattern:

\((a)-/[:KNOWS \mid INHERITS \mid LOVES]\rightarrow(\(b\))\)

Expanding the macro:

\((a)\ (\ ()\rightarrow(_e_1_s)-[_e_1:KNOWS]\rightarrow(_e_1_t)\)
  \mid (\ ()\rightarrow(_e_1_s)-[_e_1:A]\rightarrow(_e_1_t)\)
  \mid (\ ()\rightarrow(_e_1_s)-[_e_1:B]\rightarrow(_e_1_t)\)
  \mid (\ ()\rightarrow(_e_1_s)-[_e_1:C]\rightarrow(_e_1_t)\)
  \mid (\ ()\rightarrow(_e_1_s)-[_e_1:D]\rightarrow(_e_1_t)\)
  \mid (\ ()\rightarrow(_e_1_s)-[_e_1:LOVES]\rightarrow(_e_1_t)\)
\)\(\rightarrow(\(b\))\)

Variables \(x\) and \(y\) need to be replaced according to the implicit variable assignment rules in order for the macro expansion to be hygienic.

These same rules are at the same time used to assign all other missing variables, resulting in:

\((a)\ (\ ()\rightarrow(_e_1_s)-[_e_1:KNOWS]\rightarrow(_e_1_t)\)
  \mid (\ ()\rightarrow(_e_1_s)-[_e_1:A\mid B\mid C\mid D]\rightarrow(_e_1_t)\)
  \mid (\ ()\rightarrow(_e_1_s)-[_e_1:LOVES]\rightarrow(_e_1_t)\)
\)\(\rightarrow(\(b\))\)

This is equivalent to:

\((a)\ (\ ()\rightarrow(_e_1_s)-[_e_1:KNOWS]\rightarrow(_e_1_t)\)
  \mid (\ ()\rightarrow(_e_1_s)-[_e_1:A]\rightarrow(_e_1_t)\)
  \mid (\ ()\rightarrow(_e_1_s)-[_e_1:B]\rightarrow(_e_1_t)\)
  \mid (\ ()\rightarrow(_e_1_s)-[_e_1:C]\rightarrow(_e_1_t)\)
  \mid (\ ()\rightarrow(_e_1_s)-[_e_1:D]\rightarrow(_e_1_t)\)
  \mid (\ ()\rightarrow(_e_1_s)-[_e_1:LOVES]\rightarrow(_e_1_t)\)
\)\(\rightarrow(\(b\))\)
More complex example

Assume we have the macro defined:

PATH CO_AUTHORED = (x) - [:WROTE] -> (y) < - [:WROTE] - (z)

Now assume we have the pattern:

(a) -/ CO_AUTHORED /- (b)

This is rewritten to:

(a) -/ CO_AUTHORED /-> (b) | (a)<-/ CO_AUTHORED /-(b)

In an analogous manner to the previous example, we expand the macro, assign an implicit variable \_f to all instances of the path macro invocation, and then assign subscripted variations of \_f to the corresponding patterns within the expanded form of the path pattern union:

(a) ((\_f\_1) - [\_f\_2 : WROTE] - (\_f\_3) - [\_f\_4 : WROTE] - (\_f\_5)) (b) // operand 1

| (a) ((\_f\_1) - [\_f\_2 : WROTE] - (\_f\_3) - [\_f\_4 : WROTE] - (\_f\_5)) (b) // operand 2
More complex example

The first (resp. second) row is identical for both operands. In this instance, we would expect that during static analysis, the redundant union would be removed (so that only one of the operands is ever evaluated, not both), as illustrated in Example A.