TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institute für Theoretische Informatik, Wissensbasierte Systeme

Beantwortung von SPARQL-Abfragen mit der Regel-Engine Nemo

Martin Voigt

Geboren am: 20.08.1997 in Radebeul
Studiengang: Diplom Informatik
Matrikelnummer: 4607543
Immatrikulationsjahr: 2016

Diplomarbeit

zur Erlangung des akademischen Grades

Diplom-Informatiker (Dipl.-Inf.)

Gutachter
Prof. Dr. Markus Krötzsch

Betreuer
Dipl.-Math. Maximilian Marx

Eingereicht am: 23. Januar 2025

Zusammenfassung

Nemo ist eine Datalog-Engine welche spezielle Unterstützung für RDF-Daten bietet. Es ist interessant zu betrachten, inwieweit sich SPARQL-Queries mit dieser Engine umsetzen lassen.

Dazu werden in dieser Arbeit die einzelnen Features der SPARQL-Query-Sprache betrachtet und vorgestellt, wie SPARQL-Queries zu Nemo-Programme übersetzt werden können. Die Implementierung der Übersetzung ist mittels einer dafür entwickelten Nemo Template Language erfolgt. In einer Evaluation wird gezeigt, dass die Implementierung in vielen Fällen SPARQL 1.1 Standardkonform ist und auch bis RDF Graphen von über 3 Milliarden Tripels skaliert.

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich das vorliegende Dokument mit dem Titel *Beantwortung von SPARQL-Abfragen mit der Regel-Engine Nemo* selbstständig und ohne unzulässige Hilfe Dritter verfasst habe. Es wurden keine anderen als die in diesem Dokument angegebenen Hilfsmittel und Quellen benutzt. Die wörtlichen und sinngemäß übernommenen Zitate habe ich als solche kenntlich gemacht. Es waren keine weiteren Personen an der geistigen Herstellung des vorliegenden Dokumentes beteiligt. Mir ist bekannt, dass die Nichteinhaltung dieser Erklärung zum nachträglichen Entzug des Hochschulabschlusses führen kann.

Dresden, 23. Januar 2025

Martin Voigt

Inhaltsverzeichnis

1 Einleitung

In diesem Kapitel werden die Grundlagen für diese Arbeit gelegt. Dies betrifft zum einen die Arbeit auf organisatorischer Ebene mit Zielstellung (Abschnitt 1.1) und Handreichungen zum Umgang mit dem Text (Abschnitt 1.2), und zum anderen die inhaltliche Ebene mit Einführungen zu RDF (Abschnitt 1.3), SPARQL (Abschnitt 1.4) und Datalog (Abschnitt 1.5). Außerdem werden in Abschnitt 1.6 verwandte Arbeiten erwähnt.

1.1 Zielstellung

Wesentliche Teile der Zielstellung dieser Arbeit ergeben sich aus der Aufgabenstellung. Grundsätzliches Ziel ist es, SPARQL-Queries (Abschnitt 1.4) in Nemo-Programme (Unterabschnitt 1.5.2) umzuwandeln, welche den gleichen Output erzeugen.

Konkret werden die einzelnen Punkte der Aufgabenstellung hier zusammengefasst und jeweils mit der konkreten Umsetzung in Verbindung gebracht:

1. SPARQL-Anfragen erzeugen eine n-stellige Tabelle. Das erzeugte Nemo-Programm soll ein Prädikat haben, das dieselben Tupel enthält wie die Zeilen dieser Tabelle.

   - In der Arbeit wird gezeigt, wie sich aus theoretischer Sicht ein Zusammenhang zwischen den Solution Mappings, welche im Standard das Ergebnis einer SPARQL-Query bilden, und Datalog Prädikaten, herstellen lässt. Besonders im Vordergrund steht dies in Unterabschnitt 1.5.1.

2. Die Arbeit soll eine Teilmenge von SPARQL-Anfragen bestimmen, für die das korrekt möglich ist, eventuell unter bestimmten Annahmen, z.B., dass manche von Nemos eingebauten Funktionen noch minimal angepasst werden, um genau den SPARQL-Standard umzusetzen.

   - Durch eine konkrete Implementierung konnte bis auf einige Details gezeigt werden, dass unter der Annahme, dass Nemo alle SPARQL Funktionen und Aggregationen implementiert und diese Standard konform sind, alle SPARQL 1.1 Query-Language-Anfragen (ohne Berücksichtigung optionaler Features) in Nemo übersetzbar sind. Die wichtigsten Details, für welche dies nicht gezeigt wurde, sind zum einen die `FROM` Clause und Graph Management Features und zum anderen das Variablen-Scoping der `EXISTS` Expression (Unterabschnitt 3.4.9), welches einen veränderten grundlegenden Ansatz erfordert. Es ist nicht zu erwarten, dass so ein Ansatz unmöglich ist.

- Für die reale Implementierung und verwendete Nemo Version wurde eine Anstrengung unternommen, Abweichungen zum SPARQL Standard zu finden und in einigen Fällen, trotz Abweichungen der Nemo Implementierung vom SPARQL Standard eine konforme Übersetzung durchzuführen. Diese Abweichungen werden an den entsprechenden Stellen dieser Arbeit erwähnt. In der real verwendeten Nemo Version lassen sich, trotz Turing-Vollständigkeit von Nemo, einige SPARQL-Queries grundsätzlich nicht übersetzen. Durch die fehlenden Funktionen `StrLang()`, welche aus gegebenen Zeichenketten eine Zeichenkette mit Language Tag erstellt, und `StrDt()`, welche einen Literal mit Datentyp erstellt, ist es z.B. unmöglich, einige Literals im Ergebnis zu erzeugen, wenn diese nicht bereits in der Eingabe vorkommen. Nemo befindet sich in aktiver Entwicklung und es werden im Folge dieser Arbeit Issues zu entsprechenden Abweichungen vom SPARQL Standard angelegt. Es ist zu erwarten, dass die Annahme, dass Nemo alle Funktionen SPARQL konform unterstützt, in Zukunft mehr und mehr berechtigt ist.

3. Es soll eine prototypische Implementierung erstellt werden, z.B. in Rust unter Verwendung der Bibliothek Oxigraph.

- Es wurde eine prototypische Implementierung in Rust erstellt (siehe Kapitel 3). Dabei wurde die spargebra Bibliothek zum Umgang mit SPARQL-Queries verwendet, die für die Oxigraph Bibliothek entwickelt wurde und von ihr zum Transformieren einer SPARQL-Query in SPARQL-Algebra verwendet wird.

4. Die Implementierung soll verwendet werden, um eine Menge realer Anfragen zu übersetzen, z.B. die Beispielqueries des Wikidata SPARQL Endpoints oder Queries aus bekannten SPARQL-Benchmarks.

- Es wurde eine Evaluation auf den Wikidata Beispielqueries und der Offiziellen w3c SPARQL 1.1 Test Suite durchgeführt. (Kapitel 4)

In der Aufgabenstellung sind zwei Tiefen der Implementierung vorgeschlagen:

- **Minimale Lösung:** Übersetzung von Basic Graph Patterns und Property Paths
- **Umfangreiche Arbeit:** Minimale Lösung mit zusätzlicher Berücksichtigung von `FILTER`, Subqueries und Aggregaten

Es wurde die umfangreiche Arbeit umgesetzt und, wie bereits erwähnt, auch alle weiteren Features des SPARQL 1.1 Standards (`ASK` Query, `DESCRIBE` Query, `CONSTRUCT` Query, Query mit/ohne `DISTINCT` oder `REDUCED`, `UNION`, `OPTIONAL`, `EXTEND`, `MINUS`, `VALUES`, `LIMIT`, `OFFSET`, `ORDER BY`, was sich als besonders herausfordernd gezeigt hat, und Expressions inklusive `In`, `Exists`, `Bound`, `If` und `Coalesce`) berücksichtigt. Um die Menge an Features in der vorgegebenen Zeit umsetzen zu können, wurde eine Nemo Template Language mittels Rust-Makros implementiert (siehe Abschnitt 2.2).

Der Mehrwert einer SPARQL Implementierung in Datalog ist zum einen darin zu sehen, dass zusätzliches logisches Schließen vor Ausführung der Query, wie bei in Datalog umsetzbaren SPARQL Entailment Regimes, oder nach Ausführung der Query auf den Ergebnissen ohne weitere Programme möglich wird.

Außerdem können durch die Transformation von SPARQL zu Datalog auch SPARQL-Queries, wie eine Regel, Teil des logischen Schließens in Datalog werden. Für diesen Anwendungsfall ist eine SPARQL-Query Implementierung ohne stratifizierte Negation und Aggregation (siehe Unterabschnitt 1.5.2) nützlich. Deswegen wurde zusätzlich zur Aufgabenstellung bei der Implementierung darauf geachtet, nach Möglichkeit eine Implementierung der Features zu finden, welche nicht zu einem nichtstratifizierten Programm führen, wenn der Eingabegraph

vom Ergebnis der Query abhängt. Ein sehr grober schematischer Aufbau eines solchen Datalogprogramms wird in Abbildung 1.1 dargestellt.

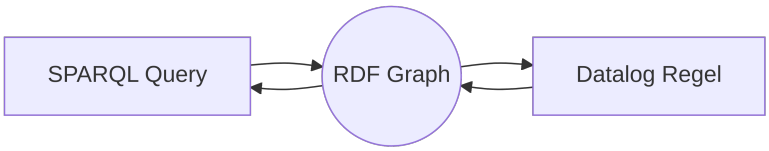

Abbildung 1.1: Rekursive SPARQL Query und Datalog Regel hängen vom Graph ab und liefern Ergebnisse für den Graphen

1.2 Umgang mit dieser Arbeit

Innerhalb dieser Arbeit werden einige Begriffe definiert und in anderen Kapiteln verwendet. Damit deutlich wird, dass es sich jeweils um einen konkret definierten Begriff handelt, sind diese Begriffe, wie im Beispiel „*repräsentiert* ↗ “, durch einen Pfeil markiert. Der Pfeil verlinkt zur Definition. Unterstützt der verwendete PDF-Viewer Link-Preview, kann somit, durch das Halten der Maus über den Pfeil, die Definition angezeigt werden. Alternativ kann der Begriff im Anhang unter „Begriffsverzeichnis“ (Abschnitt 6.1) nachgeschlagen werden. Generell werden in dieser Arbeit häufig andere Kapitel referenziert. Dies zeigt wichtige Voraussetzungen für ein Kapitel auf und kann auch als Gedächtnisstütze dienen. Bei linearem Lesen der Kapitel sollten jedoch alle relevanten Grundlagen für ein Kapitel bereits bekannt sein.

1.3 RDF

Ein *Triple-Graph* ↗ ist eine Menge an Triples.

Typischerweise wird ein Graph durch seine Kanten und Knoten definiert. Ausgehend von der oben genannten Definition sind unterschiedliche Definitionen für die Knoten eines Triple-Graphen denkbar. SPARQL definiert beispielsweise `nodes(G)` eines Triple-Graphen `G`:

```
nodes(G) = { n | n is an RDF term that is used as a subject or
object of a triple of G}
```

Ein „RDF term“ ist dabei eine IRI[1], ein Literal[2] oder ein BNode[3].

Einige Notationen im Zusammenhang mit RDF erlauben eine verkürzte Notation von IRIs indem ein wiederkehrendes Präfix einen kürzeren Namen bekommt. Im Laufe der Arbeit werden folgende RDF-Präfixe teilweise, ohne auf deren Definition einzugehen, verwendet:

Prefix	Namespace-IRI
`ex:`	https://example.org/
`xsd:`	http://www.w3.org/2001/XMLSchema#

[1] IRIs sind innerhalb dieser Arbeit gleichbedeutend mit allgemein bekannten URLs. IRIs unterstützen jedoch zusätzliche Features, wie Unicode-Symbole.
[2] ein konkreter Wert eines Datentyps
[3] Ein „Blanker“ Knoten ohne Bedeutung außerhalb des aktuellen Graphen

Nach dieser Definition steht beispielsweise `ex:abc` für die IRI `https://example.org/abc`.

In RDF heißt die erste Stelle eines Triples im Graph „subject“, die zweite „predicate“ und die dritte „object“. Dies ist angelehnt an die englische Grammatik. Hier ein deutsches Beispiel: „Mein Stift hat die Farbe Blau.“, dies kann als Triple („Mein Stift“, „hat die Farbe“, „Blau“) ausgedrückt werden.

1.4 SPARQL

Eine wichtige Query-Sprache für RDF ist SPARQL. SPARQL ist in seiner grundlegenden Struktur, z. B. Verwendung von Keywords wie `SELECT` oder `ORDER BY`, ähnlich zu SQL, der allgemein bekannten Query Sprache für relationale Datenbanken. Die konkreten Features sind jedoch angepasst auf Graphdatenbanken, welche auf RDF (Abschnitt 1.3) basieren. So gibt es Features, welche ein gegebenes Graph Pattern in einem Graphen finden oder Features zum Finden von Knoten, welche durch einen Pfad mit bestimmter Struktur verbunden sind. In dieser Arbeit wird sich auf den SPARQL 1.1 Standard [7] bezogen. Der Standard beschreibt außer einer Query-Syntax auch die SPARQL Algebra, welche eine Zwischenrepräsentation bietet, die es erlaubt, existierende Query-Parser und Algebra-Transformationen in einer Datenbankimplementierung zu verwenden.

Das Ergebnis einer SPARQL-Query ist laut Standard eine (möglicherweise ungeordnete) Sequenz von „solution mappings“. Ein Solution Mapping ist dabei eine Funktion, welche für eine gegebene Variable aus der Query einen RDF-Wert ergibt. Ein Solution Mapping muss dabei nicht für jede Variable der Query einen Wert ergeben. Für einige Variablen kann es nicht definiert sein. Diese Definition gilt auch für das Ergebnis von vielen Operationen der SPARQL-Algebra.

Im Rahmen dieser Arbeit ist an einigen Stellen im Zusammenhang mit SPARQL-Queries von einem *Query-Ergebnis*↗ die Rede. Dies ist hier als eine Menge von Tupels definiert, mit der Intuition, dass sich das Ergebnis als Tabelle darstellen lässt und jede Zeile der Tabelle ein Tupel ist. Es ist jedoch nicht trivial, Ergebnisse von SPARQL-Queries oder Subqueries als Tupel darzustellen. Im Laufe dieser Arbeit wird eine konkrete Umsetzung vorgestellt. Ein Überblick wird in Unterabschnitt 1.5.1 gegeben. Dort wird der Zusammenhang zwischen beiden Ergebnisdefinitionen hergestellt.

1.5 Datalog

Datalog ist eine Regelsprache, bei der aus bereits bekannten Fakten durch Regeln neue Fakten abgeleitet werden können. Die hier vorgestellte Datalog-Definition folgt der von Ceri und anderen [3] vorgestellten Beweis-theoretischen Sichtweise. Sie wurde in dieser Arbeit umfangreich angepasst, um den Fokus auf relationale Betrachtungen, welche im Zusammenhang mit SPARQL relevant sind, zu legen.

Fakten in Datalog sind n-Tupel, die einem n-stelligen Prädikat zugeordnet sind. Syntaktisch wird dabei das Prädikat vor das Tupel geschrieben. So ist beispielsweise `parent(ann, dorothy)`, das Tupel (*ann*, *dorthy*), welches dem zweistelligen Prädikat `parent` zugeordnet wird. Fakten können durch eine Funktion I_i, welche jedem Prädikat p die Menge der zugeordneten Tupels $I_i(p)$ zuordnet, beschrieben werden. Explizit im Datalog-Programm geschriebene initiale Fakten werden durch die Funktion I_0 beschrieben.

Datalog Regeln haben folgende syntaktische Form:

$$p_{0x}(t_1, \ldots, t_x) \ :- \ p_{1y}(t_1, \ldots, t_y), \ldots, p_{mz}(t_1, \ldots, t_z)$$

p_{jn} steht für ein n-stelliges Prädikat. Die Terme Terme $t_1, \ldots, t_n$ können Konstanten oder Variablen sein. Variablen sind syntaktisch durch ein ?-Präfix markiert. Es wird davon gesprochen, dass in einer Regel, eine Konstante oder Variable an die Position eines Prädikats „*gebunden*↗ " wird. So wird z.B. t_1 an die erste Position des Prädikats gebunden. Der Teil vor dem „ :-"-Symbol wird als *Regel-Kopf*↗ und der Teil danach als *Regel-Körper*↗ bezeichnet.

Wie durch eine Regel neue Fakten abgeleitet werden können, kann durch Funktionen I_i, wie oben eingeführt, beschrieben werden. Ein Tupel t ist in $I_i(p)$ genau dann, wenn t in $I_{i-1}(p)$ ist oder es gibt eine Regel und Variablenersetzung θ mit $p(t_1, \ldots, t_x)$ als *Regel-Kopf*↗, $t = (\theta(t_1), \ldots, \theta(t_x))$ und für jeden Teil des Regel-Körpers $p_{jn}(t_1, \ldots, t_n)$ gilt $(\theta(t_1), \ldots, \theta(t_n)) \in I_{i-1}(p_{jn})$. Eine Variablenersetzung erhält dabei alle nicht-Variablen. Für alle nicht-Variablen k gilt also $\theta(k) = k$. Es gibt eine Safety-Bedingung, welche erfüllt ist, wenn jede Variable, die im Regel-Kopf vorkommt, auch im Regel-Körper gebunden wird. Die Safety-Bedingung stellt sicher, dass keine Fakten mit beliebigen Konstanten, welche unabhängig vom Datalog-Programm sind, abgeleitet werden können.

Eine einfache Regel ist beispielsweise „dorothy hat dieselben Eltern wie hilary":

```
parent(?Parent, hilary) :- parent(?Parent, dorothy)
```

Es ist bekannt [3], dass sich in Datalog jeder ableitbare Fakt in endlich vielen Schritten ableiten lässt. Es gibt also ein q mit $I_i = I_{i+1} = I$ für $i > q$, wobei I_i die oben definierte Funktion ist, welche einem Prädikat alle bis zum i-ten Schritt abgeleiteten Tupel zuordnet. I wird „*Lösung*↗ " des Datalog Programms genannt. Ist I die *Lösung*↗ eines Programms und p ein Prädikat des Programms, so kann dies durch „*p repräsentiert*↗ $I(p)$" oder „*p steht für*↗ $I(p)$" ausgedrückt werden. Das bedeutet, ein Prädikat *repräsentiert*↗ eine Relation in einem Datalog-Programm. Außerdem wird $I(p)$ das „*Ergebnis*↗ " von p genannt.

Gibt es für eine Regel eine Variablenersetzung, sodass die dazugehörigen Tupel des Regel-Körpers sich im entsprechenden *Ergebnis*↗ des Prädikats befinden, so wird dies dadurch beschrieben, dass die Regel „*zum Tragen*↗ " kommt. Ist dies nicht der Fall, wird also die Regel nicht angewendet, wird dies durch „Die Regel kommt nicht *zum Tragen*↗ " beschrieben.

In der Praxis gibt es zahlreiche Erweiterungen von Datalog, diese erhöhen mitunter die Ausdrucksstärke und führen häufig zu anderen Eigenschaften. Erlaubt eine Erweiterung beispielsweise das Erzeugen neuer Konstanten in rekursiven Regeln, ist im Allgemeinen eine endliche Ableitung aller Fakten nicht garantiert und eine, wie oben definierte Lösung des Datalog Programms, existiert in einigen Fällen nicht. Dennoch sind die in diesem Kapitel gegebenen Definitionen, unter Berücksichtigung einiger zusätzlicher Fälle, auch für die in dieser Arbeit betrachteten Erweiterungen relevant.

1.5.1 Datalog mit Prädikat-Attributen

Es ist möglich, jeder Position eines Datalog-Prädikats ein Attribut zuzuweisen. Dies beeinflusst nicht die *Lösung*↗ eines Datalog-Programms. Dafür vereinfacht es das Generieren von Regeln, besonders, wenn zum Zeitpunkt des Entwickelns der Regel die Arität oder konkrete Bedeutung der einzelnen Positionen eines Prädikates unbekannt sind.

Für gegebene Prädikate kann man eine Datalog-Variable für jedes Attribut festlegen und die Prädikate *gebunden*↗ an die jeweiligen Variablen in einem *Regel-Körper*↗ zusammenführen.

Dies definiert eine natürliche Art, Prädikate in einer Regel zu verknüpfen und entspricht einem `Natural Join` in SQL.

Sei P die Menge der Positionen (Indices) eines Prädikats p, mit $|P|$ der Arität von p. Sei A eine Menge von Attributen und W eine Menge von Werten. Ein n-tupel kann definiert werden als eine Funktion, deren Domain eine n-elementige Indexmenge ist. Nach dieser Definition ist ein Tupel im Ergebnis von p eine Funktion $P \to W$ und die Attributzuweisung für p eine Funktion $P \to A$. Schreibt man das Ergebnis von p als Tabelle mit jedem Tupel in einer Zeile, so kann die Attributzuweisung den Header der Tabelle bilden.

Prädikate mit Attributen ermöglichen es, Solution Mappings aus SPARQL (Abschnitt 1.4) darzustellen. Das *Query-Ergebnis*↗ einer SPARQL-Query lässt sich wie ein Prädikat mit Attributen als Tabelle mit Header darstellen, wobei jede Zeile der Tabelle ein Solution Mapping darstellt. Formal ist ein Solution Mapping eine Funktion, z.B. μ, von bestimmten SPARQL-Variablen (V) zu RDF terms (W, siehe Abschnitt 1.3). Hat man ein Datalog Prädikat p (mit Indexmenge P) mit Tupel $t : P \to W$ im *Ergebnis*↗ und p hat die Atributzuweiseung $a : P \to V$, so kann daraus ein Solution Mapping definiert werden: $\mu = t \circ a^{-1}$ (also $\mu(x) = t(a^{-1}(x))$). Eine Attributzuweisung wird hier als invertierbar angenommen, das bedeutet, jedes Attribut darf nur einmal verwendet werden.[4]

Erweitert man die Definition für ein einzelnes Solution Mapping auf das gesamte *Ergebnis*↗ eines Prädikats, korrespondiert das *Ergebnis*↗ eines Prädikats zu einer Solution Sequence ohne Reihenfolge. Der SPARQL-Standard [7] definiert „A solution sequence is a list of solutions, possibly unordered“, wobei jedes Solution Mapping hier immer für alle Variablen im Image der Attributzuweisung des Prädikats definiert ist. Wie andere SPARQL Ergebnisse dargestellt werden können, wird an den entsprechenden Stellen dieser Arbeit erläutert. Dies erfolgt insbesondere in Abschnitt 2.3 für Solution Mappings mit Reihenfolge, im Abschnitt 3.1 für Solution Mappings mit unterschiedlichen Domains und im Abschnitt 3.8 für SPARQL-Ergebnisse die keine Solution Sequences sind.

1.5.2 Nemo

Eine konkrete Datalog-Implementierung ist die Regel Engine mit dem Namen „Nemo“ [4]. Ein Ziel von Nemo ist es, RDF-Knoten, wie Literale unterschiedlicher Datentypen, IRIs und BNodes, sowie SPARQL Funktionen direkt zu unterstützen. Dies macht Nemo besonders attraktiv für eine SPARQL zu Datalog Übersetzung.

Einige Features, um welche Datalog in Nemo erweitert wird, sind an dieser Stelle nennenswert.

Nemo unterstützt Constraints, wie folgende Beispielregel zeigt:

```
q(1, "1") .
p(?a) :- q(?a, ?b), ?b = STR(?a) .
```

Der Constraint `?b = STR(?a)` schränkt z.B. das Ergebnis auf solche `?a` Werte ein, deren String Repräsentation der dazugehörige `?b` Wert ist. Dabei ist es erlaubt, den Funktionsaufruf direkt in das Prädikat `q` zu schreiben:

```
p(?a) :- q(?a, STR(?a)) .
```

[4] Eine Verallgemeinerung, bei der Attribute mehrfach verwendet werden, aber alle Tupel den gleichen Wert an den entsprechenden Positionen haben, wäre möglich jedoch nicht praktisch sinnvoll. Die Verwendete Bibliothek spargebra sowie die Blazegraph Datenbank erlauben beispielsweise keine Variable doppelt im SELECT Pattern.

Nemo unterstützt stratifizierte Negation mit dem ~ Symbol, wie folgendes Beispiel zeigt:

```
a(1) . a(2) . b(1) .
c(?x) :- a(?x), ~b(?x) .
```

Das Nemo-Programm ergibt nur `c(2)` und nicht `c(1)`, da es für `c(1)` ein entsprechendes `b(1)` gibt. Der Begriff „stratifiziert" bedeutet dabei eine Einschränkung, welche besagt, dass alle Prädikate des Programms in eine Sequenz von „Schichten" eingeteilt werden können. Hängt ein Prädikat negativ von einem anderen Prädikat ab, so muss sich dieses in eine höhere Schicht einteilen lassen. Im Beispiel hängt `c` negativ von `b` ab und muss somit in einer höheren Schicht sein. Die Stratifizierungsbedingung schließt nicht sinnvolle Datalog Programme, bei denen Prädikate zyklisch negativ von voneinander abhängen, wie das folgende, aus:

```
p(?x) :- ~p(?x) .
```

Nach der Stratifizierungsbedingung müsste `p` einer höheren Schicht als es selbst ist zugeordnet werden, was unmöglich ist. Positive, zyklische Abhängigkeiten innerhalb einer Schicht sind erlaubt. Beim Schreiben oder Generieren eines Nemo-Programms ist zu beachten, dass die Stratifizierungsbedingung erfüllt ist.

Nemo unterstützt Aggregation, wie folgendes Beispiel zeigt:

```
a(1) . a(2) .
b(#sum(?x)) :- a(?x) .
```

Das Programm ergibt `b(3)`. Es ist zu beachten, dass Aggregation, ähnlich wie Negation, eine höhere Schicht fordert. Im Beispiel muss `b` einer höheren Schicht als `a` zugewiesen sein.

Nemo unterstützt existentielle Regeln durch Variablen-Präfix `!`, wie folgendes Beispiel zeigt:

```
next(1, 2) . next(2, 3) . next(3, 1) .
next(?y, !z) :- next(?x, ?y) .
```

Die Regel besagt, dass für jedes `?y`, welches an zweiter Stelle im `next` Prädikat vorkommt, auch ein Tupel mit `?y` an erster Stelle existieren muss. Im Beispiel ist dies bereits für das `next` Prädikat erfüllt. Wäre es für einen Wert nicht erfüllt, würde Nemo einen neuen Wert `!z` als BNode generieren. Dies würde eine endlose `next`-Kette und damit ein nicht terminierendes Nemo Programm bewirken. Im Allgemeinen ist schwierig festzustellen, ob eine solche existentielle Bedingung bereits durch andere Regeln erfüllt wird. In der Praxis kann es vorkommen, dass Nemo durch eine existentielle Regel einen BNode generiert, obwohl zusätzlich durch eine andere Regel später auch ein entsprechender Wert existieren wird. Das Verhallten hängt also von der internen Anwendungsreihenfolge der Regeln ab.

Zum Zeitpunkt der dieser Arbeit (2025) ist Nemo in aktiver Entwicklung mit häufigen Änderungen. Grundlage für diese Arbeit ist eine Version aus dem Juli 2024. Darauf aufbauend wurden die `UCASE` und `LCASE` Funktionen durch vertauschen korrigiert und das behandeln von Fehlern bei den `SUBSTR` und `SUBSTRING` Funktionen an den SPARQL standard angepasst. Die resultierende Version wird in dieser Arbeit, als Abhängigkeit für die Implementierung und für die Tests, Verwendet. Sie ist unter

https://github.com/knowsys/nemo/tree/36ca7d40295203099db04a501642b4686b2f2009

einsehbar.

1.6 Existierende SPARQL zu Datalog Implementierungen

Die Idee, SPARQL in Datalog zu übersetzen existiert bereits, seit es SPARQL gibt. SPARQL wurde 2008 in der Version 1.0 standardisiert. Bereits 2007 stellte Simon Schenk in einem Konferenzpaper eine Definition der SPARQL Semantik, basierend auf einer Übersetzung nach Datalog, vor [6]. Ein Paper von Renzo Angles und Claudio Gutierrez aus dieser Zeit analysiert die Ausdrucksmächtigkeit von SPARQL und verwendet dafür eine Übersetzung nach Datalog [1]. Beide Paper enthalten keine reale Übersetzungsimplementierung. Außerdem ist anzumerken, dass sich der SPARQL 1.0 Standard signifikant leichter in Datalog ausdrücken lässt, da viele Features erst im SPARQL 1.1 Standard hinzugekommen sind. Beispielsweise erzeugen die Übersetzungen null Werte für den `OPTIONAL` Query-Pattern, welche allerdings für die restliche Query keine größere Rolle spielen, da es im SPARQL 1.0 Standard z.B. noch keine Subqueries gibt.

Eine aktuelle Übersetzung von SPARQL nach Datalog wird von Angles und Anderen mit dem SparqlLog System vorgestellt [2]. Die Übersetzung findet für die Datalog-Engine Vadalog statt. Es wird Multiplizität von Ergebnissen unterstützt, indem durch existentielle Regeln in Datalog IDs, welche die Ergebnisse unterscheiden, generiert werden. Für die Unterstützung von `ORDER BY` gereift die Implementierung auf die bereits in Vadalog verfügbare Sortierfunktion zurück, diese ist jedoch nicht SPARQL konform. SparqlLog implementiert einige Features des SPARQL 1.1 Standards nicht. Insbesondere sind hier `CONSTRUCT` Query, `DESCRIBE` Query, `BIND`, `VALUES`, `HAVING`, `FILTER EXISTS` / `FILTER NOT EXISTS`, `Coalesce`, `IN`, `NOT IN`, Group Graph Pattern, Subqueries und `FROM` zu nennen. Weitere Features sind weder als implementiert noch als nicht implementiert aufgelistet. Die Implementierungen der unterstützten Features ähneln sich teilweise denen in dieser Arbeit vorgestellten. So verwendet SparqlLog beispielsweise auch[5] ein dreistelliges Prädikat, um mit Unbound-Werten umzugehen. In SparqlLog ist dieses jedoch, ohne auf nähere Details einzugehen, ein einmal berechnetes Prädikat, während in dieser Arbeit aufgrund der unterstützen Features innerhalb der Übersetzung mitunter mehrere Versionen eines äquivalenten Prädikats benötigt werden.

[5] siehe Unterabschnitt 3.7.3 in dieser Arbeit

2 Design des SPARQL zu Nemo Übersetzungstools

In Kapitel 1 wurde bereits das Ziel, ein Programm zu entwickeln, welches SPARQL-Queries zu Nemo Programmen transformiert, mit den relevanten Grundlagen vorgestellt. In diesem Kapitel werden nun grundlegende Designentscheidungen für ein solches Programm getroffen. Das Programm wird in der Programmiersprache Rust entwickelt. Die Betrachtung der Ein- und Ausgaben findet in Abschnitt 2.1 statt. Eine wichtige Designentscheidung besteht darin, die Nemo-Regeln mittels einer Nemo Template Language zu generieren. Diese wird in Abschnitt 2.2 beschrieben. Außerdem werden durch die Implementierung der Nemo Template Language wichtige Grundlagen gelegt, wie die vom Programm verarbeiteten Datenstrukturen aussehen. Diese werden ebenfalls in Abschnitt 2.2 vorgestellt. In Abschnitt 2.3 wird die Entscheidung darüber getroffen, wie die Unterschiedlichen SPARQL Ergebnistypen, wie Ergebnisse mit Reihenfolge, in Nemo dargestellt werden.

Weiterhin schließt das Design ein, wie der Übersetzungsvorgang grundsätzlich abläuft, dies wird in Abschnitt 2.4 vorgestellt, und wie die Tests stattfinden, was in Abschnitt 2.5 vorgestellt wird.

2.1 Grundlegendes Setup des Programms

Das Programm für die Übersetzung einer SPARQL-Query zu einem Nemo Programm verfügt nicht über Command-Line-Argumente oder ein komplexeres Userinterface. Wichtige Eingaben erfolgen direkt im Quelltext, indem die Funktion `translate(query_str)` aufgerufen wird. Allerdings gibt es die Möglichkeit, die SPARQL-Query aus stdin zu lesen und das Nemo Programm auf stdout zu schreiben, sodass eine einfache Interaktion mit anderen Programmen möglich wird. Außerdem gibt es eine Rust-Testsuite, welche Beispielqueries übersetzt und die resultierenden Nemo-Programme mittels Nemo auf testspezifischen Daten evaluiert und mit den jeweils erwarteten Ergebnissen vergleicht. (siehe Abschnitt 2.5)

2.2 Nemo Template Language

Die SPARQL zu Nemo Konvertierung ist, sehr grob betrachtet, eine String Transformation. Eine gegebener SPARQL-Query-String wird zu einem Nemo-Programmfragment-String konver-

tiert. Eine initiale Implementierung verwendete direkte die verfügbaren String-Manipulations-Features der Rust-Programmiersprache. Dies erschwert jedoch das Verständnis der Übersetzung stark, was die Anzahl der unterstützten Features aufgrund der Zeitvorgaben reduziert hätte.

Die „Nemo Template Language“ wurde im Rahmen dieser Arbeit dazu entwickelt, Nemo-Programme, basierend auf Nemo-Programm-Templates, zu generieren. Sie stellt im Wesentlichen eine Abstraktion für die benötigten String Manipulationen beim Generieren von Nemo-Programmen dar. Dazu wird mittels Rust-Makros Rust-Programmcode generiert, welcher eine interne Repräsentation eines Nemo-Programms erzeugt, welche dann zum Generieren eines Nemo-Programm-Strings verwendet wird.

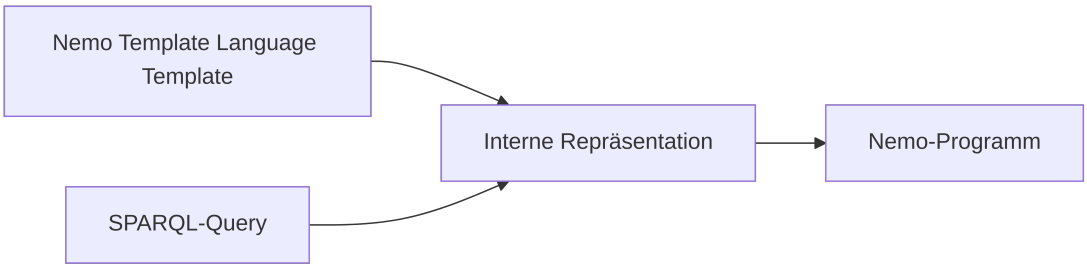

Abbildung 2.1: Transformation von SPARQL-Query zu Nemo-Programm

Die interne Repräsentation kann auch ohne Makros manipuliert werden. Dies ermöglicht es, weniger häufig verwendete Features zu unterstützen, für die sich keine Implementierung im Makro lohnt. Es wird eine nahezu nahtlose Verwendung von Rust-Makros mit manuell erstellten Teilen der internen Repräsentation unterstützt. Beispielsweise sind die grundlegenden Operatoren für intern repräsentierte Nemo Variablen überladen und eine Rust-Expression kann direkt innerhalb eines Nemo Templates verwendet werden. Außerdem gibt es zahlreiche Implementierungen des `From` Traits in Rust, welcher generische Implementierungen der Makros, inklusiv Unterstützung für Built-in Rust Typen, ermöglicht. Loops und Branches in der Template Language werden durch die Verwendung der entsprechenden Kontrollstrukturen in Rust umgesetzt. So können z.B. Regeln oder Regelteile unter Verwendung der Nemo Template Language in einem normalen Rust-Loop erzeugt und dann in weiteren Makroaufrufen verwendet werden.

Eine besondere Herausforderung stellt der Umgang mit Prädikaten dar, deren Arität erst während der Ausführung des Programms bekannt ist. Auf jeden Fall sind die Positionen und Arität des Ergebnisprädikats (siehe Abschnitt 1.1) ohne die SPARQL-Query nicht bekannt, da dies von den Variablen im `SELECT` in der Query abhängt. Gleiches gilt aber auch für fast alle Zwischenergebnisse, sodass nur bei einigen wenigen Regeln die Arität der Prädikate zur Zeit des Schreibens des Nemo Templates bekannt ist. Zwar ließe sich die Liste der Bindings für die Positionen jedes Prädikats mittels eines Loops in Rust generieren, dies wäre jedoch wieder nah an einer Implementierung mittels String Manipulation und würde somit die Vorteile der Nemo Template Language negieren. Stattdessen erlaubt die Nemo Template Language, die Regelstruktur in einer Nemo nahen Form zu schreiben und die konkrete Zuordnung zu den Prädikat-Positionen geschieht erst zur Übersetzungszeit. Damit dies möglich wird, betrachtet die Nemo Template Language alle Prädikate als Prädikate mit Attributen (Unterabschnitt 1.5.1), diese Attribute werden aus den Regeln automatisch von der Nemo Template Language abgeleitet.

So könnte beispielsweise eine Regel in Nemo Template Language Aussehen:

```
a(??x, ??y) :- b(??x, ??y), c(??x)
```

Dabei können Variablen mit `??` Präfix für mehrere Nemo Variablen stehen. Wie dies konkret funktioniert wird im Unterabschnitt 2.2.1 beschrieben, außerdem werden dort sowie in Unterabschnitt 2.2.4 weitere Features vorgestellt. Folgende Regel könnte aus dem Template oben resultieren:

```
a(?u, ?w, ?v) :- b(?u, ?w, ?v), c(?u, ?v) .
```

Dabei steht `??x` aus dem Template für `?u` und `?v` in Nemo, und `??y` für `?w`. Es ist zu beachten, dass im Template `??x` vor `??y` steht, dies jedoch nicht bedeuten muss, dass nicht `?w` zwischen `?u` und `?v` stehen kann. Die Nemo-Regel wird anhand des Templates und der Information, welche Attribute die Prädikate `b` und `c` haben, gebildet. Ist `a` ein neues Prädikat, werden durch die Regel die Attribute von `a` inferiert.

Außerdem ist es möglich, Eigenschaften der jeweiligen Attribute automatisch zu tracken. Eine Eigenschaft ist dabei eine universell quantifizierte notwendige Aussage, welche wahr oder falsch sein kann. Ein Beispiel für so eine Aussage wäre:

> „Alle Werte, die ein Attribut annehmen kann, sind notwendigerweise gerade Zahlen."

Es ist anzumerken, dass dies das Gegenteil folgender Aussage ist:

> „Es existiert möglicherweise ein ungerader Wert."

Es können also durch Umformung auch einige existentiell quantifizierte Aussagen unterstützt werden. Diese existentielle Sichtweise zeigt, warum diese Art von Eigenschaften Performance relevant sind: Ist es unmöglich, dass ein Wert ungerade ist, müssen die Nemo-Regeln zum Behandeln ungerader Werte nicht generiert werden, was unter Umständen die Programmlaufzeit verbessert. Aus der Definition ergibt sich, dass bei einer Regel wie

```
a(?x) :- b(?x), c(?x) .
```

die Eigenschaft für das Attribut `?x` vom Prädikat `a` gilt, wenn sie für das Attribut von `b` *oder* `c` gilt (siehe Anhang 6.3) und dass bei zwei Regeln die Eigenschaft aufgrund von Regel 1 *und* Regel 2 gelten muss, damit sie für das Prädikat gilt. Ist ein Prädikat nicht im Kopf einer Regel, *repräsentiert*↗ es keine Tupel, somit wird jede über diese Tupel universell quantifizierte Aussage wahr. Es ergibt somit Sinn, dass die getrackten Eigenschaften wahr sind, solange nicht durch eine Regel abgeleitet wird, dass sie falsch sind oder explizit die Aussage auf Falsch gesetzt wird. Für alle komplizierteren Fälle, wie Negation, Funktionsaufrufe und Operatoren in Nemo, müssen die Eigenschaften, wenn nötig, manuell gesetzt werden.

Die Implementierung der Nemo Template Language findet im Projekt des Übersetzungstools in der Datei `nemo_model.rs` statt. Da die konkrete Implementierung nur indirekt relevant zur Beantwortung der Aufgabenstellung ist, werden hier nur entscheidende Konzepte und Algorithmen vorgestellt und weniger auf ihre konkrete Implementierung eingegangen. Die Implementierung umfasst über 2500 Zeilen und ist damit etwas länger als die eigentliche Übersetzung, welche fast 2000 Zeilen umfasst. Diese Zahlen vermitteln eine ungefähre Vorstellung für den Umfang der Implementierung.

2.2.1 VarSets

VarSets wurden bereits in der Einführung (Abschnitt 2.2) als Möglichkeit zum Umgang mit Prädikaten, deren Anzahl und Reihenfolge der Positionen zum Zeitpunkt des Templateschreibens unbekannt ist, vorgestellt. Ein erweitertes Beispiel für eine Regel in Nemo Template Language:

```
a(1+u, ?v, ??x, ??y) :- b(u, ??x, ??y), c(??x, ?v)
```

Außer den VarSets `??x` und `??y` gibt es im Beispiel auch die einzelne Variablen `?v` und eine explizite Variable `u`. Dabei ist `u` zuvor in Rust durch `let u = nemo_var!(b);` definiert worden. Die mit `?` beginnenden Variablen und VarSets sind nur innerhalb des Templates gültig und müssen nicht zuvor definiert werden.

Die allgemeine Struktur der Parameter eines Prädikats in Nemo Template Language besteht aus einem Präfix von positionsbasierten Variablen, einer Mitte aus VarSets und einem Suffix aus positionsbasierten Variablen. Explizite und einzelne Variablen sind dabei positionsbasiert. Im Beispiel bedeutet dies, dass `1+u`, `u` und `?v` nur in Präfix- oder Suffix-Position vorkommen. Eine positionsbasierte Variable ist nicht zwischen VarSets erlaubt. Dadurch können die Attribute vorne und hinten aufgrund ihrer Position direkt zu einzelnen Variablen zugeordnet werden und sind nicht an der Aufteilung in VarSets beteiligt.

Die Aufteilung der nicht positionsbasiert zugeordneten Prädikatatribute auf die VarSets ist potenziell nicht trivial. Im Beispiel ist das VarSet `??y` dadurch definiert, dass ihm alle Attribute zugeordnet werden, die im Prädikat `b` vorkommen, aber nicht bereits dem VarSet `??x` zugeordnet worden sind. Solche Aussagen könnten prinzipiell kettenreaktionsartig auf andere VarSets Einfluss haben und dazu führen, dass ein Verfahren zum logischen Schließen für die Aufteilung verwendet werden müsste. Dies wurde jedoch nicht implementiert und stattdessen eine einfache Semantik für VarSets verwendet. Nach dieser gehört jedes an VarSets beteiligtes Attribut prädikatübergreifend genau einem VarSet an. Durch diese einschränkende Annahme steht jedes VarSet für einen Bereich im Venn-Diagramm, welches sich aus den Mengen der Attribute für jedes Prädikat in der Regel ergibt. Es ist zu erwähnen, dass ein Venn-Diagramm potenziell exponentiell viele Bereiche haben kann, was zu vielen VarSets in einer Regel führen kann. Der Autor eines Templates muss darauf achten, dass alle real möglichen Bereiche des Venn-Diagramms durch genau ein VarSet abgedeckt werden. Die Praxis zeigt, dass einige Regeln etliche VarSets benötigen. Es stellt jedoch kein signifikantes Problem dar, da Regeln mit vielen Prädikaten häufig eine einfachere Struktur haben, im Hinblick auf Prädikate, welche gemeinsame Attribute beinhalten. Außerdem besteht stets die Möglichkeit eines manuellen Erstellens mittels Loop in Rust.

Aus dem Template oben kann sich folgende Beispiel Nemo-Regel ergeben:

```
a(?u + 1, ?m, ?n, ?o, ?p) :- b(?u, ?n, ?o, ?p), c(?n, ?o, ?m) .
```

Die nicht positionsbasiert zugeordneten Attribute sind `?n`, `?o` und `?p` in `b` und `?n` und `?o` in `c`. Daraus ergibt sich das einfache Venn-Diagramm in Abbildung 2.2, wobei die Menge der Attribute von `c` eine Teilmenge der Attribute von `b` ist. Es ergeben sich zwei Bereiche. Die Schnittmenge `{?n, ?o}` und die Menge der Attribute von `b`, die nicht in `c` sind, `{?p}`. Betrachtet man die VarSets im Template ergibt sich das `??x` in `b` und `c` vorkommt und somit für die Schnittmenge `{?n, ?o}` steht und `??y` nur in `b` aber nicht in `c` vorkommt und somit für `{?p}` steht.

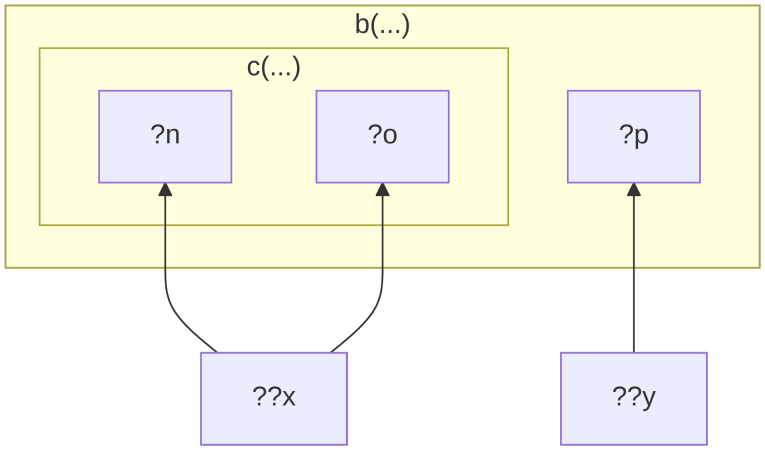

Abbildung 2.2: VarSets stehen für Bereiche im Venn-Diagramm aus den Attribut-Mengen der Prädikate

Die Zuordnung lässt sich durch Bitmanipulation implementieren. Jedem Prädikat der Regel wird eine Bit-Position zugeordnet. Für jedes Attribut werden für die Variablen, in denen es vorkommt, die Bits auf 1 gesetzt. Genauso werden für jedes VarSet die Positionen für alle Prädikate, in denen es vorkommt, auf 1 gesetzt. Dadurch ergibt sich ein u64 Bit Integer in Rust für jedes Attribut und jedes VarSet. Ein VarSet enthält dann genau alle Attribute mit der gleichen Zahl, die sich für das VarSet selbst ergibt. Gibt es zwei VarSets mit der gleichen Zahl oder für eine Zahl eines Attributs kein VarSet, führt dies zu einem Fehler zur Übersetzungszeit, also der Laufzeit des Übersetzungstools. Außerdem werden Regeln, bei denen 64 Bit nicht ausreichen, nicht unterstützt. Es ist durchaus möglich und praktisch relevant, dass ein VarSet keine Attribute enthält. Wenn das Prädikat im Kopf der Regel, im Beispiel a, bereits definiert ist, nimmt es auch am Prozess zur Auflösung der VarSets teil.

Ein Attribut kann zwar immer nur in einem VarSet sein, es ist jedoch möglich, dass es sowohl positionsbasiert ist, als auch in einem VarSet zugeordnet werden kann. Eine Regel wie

```
a(attr) :- b(??all_attrs)
```

kann somit dazu verwendet werden, ein zuvor definiertes Attribut `attr`, welches in den Attributen von b vorkommt, zu extrahieren, obwohl das Attribut sowohl explizit definiert als auch Teil des VarSets ist. Dieses grundsätzliche Verhalten, dass einzelne Variablen vollständig positionsbasiert sind und so auch für bereits anders vergebene Attribute stehen können, kann unter Umständen auch ungewollt sein. Existiert z.B. ein Prädikat p mit Attribut ?a, so ergibt das Regeltemplate

```
q(?x, ?y) :- p(?x), p(?y)
```

das Prädikat q mit den Attributen ?a und ?a und das Nemo-Programm zum Template ist

```
q(?a, ?a) :- p(?a), p(?a) .
```

da beide einzelnen Variablen an dasselbe Attribut ?a *gebunden*↗ werden, was aus dem Regeltemplate unter Umständen nicht offensichtlich ist. Bei expliziten Variablen existiert eine höhere Kontrolle. Definiert man die Variablen

```
let x = nemo_var!(x); let y = nemo_var!(y);
```

ergibt das Template

```
q(x, y) :- p(x), p(y)
```

die Nemo Regel

```
q(?x, ?y) :- p(?x), p(?y) .
```

Das Prädikat `q` hat in diesem Beispiel die Attribute `?x` und `?y`. Ähnliches gilt auch für Expressions, da diesen jeweils eine eigene explizite Variable zugeordnet ist. Im Beispiel, welches am Anfang diese Kapitels eingeführt wurde, führt die Expression `1+u` zu einem neuen Attribut `?SUM` im resultierenden Prädikat `a`. Gibt es mehrere Summationen in der Regel, bekommen zwar alle ein Attribut, welches `?SUM` heißt. Dies sind jedoch unterschiedliche Attribute und werden dann bei der Erstellung des Nemo-Programms im Kollisionsfall umbenannt.

Eine spezielle Form von VarSets sind RenameSets, diese werden mit `??*` Präfix geschrieben statt nur `??`. Es kann nur ein RenameSet pro Prädikat geben und jedes RenameSet darf nur einmal im Body einer Regel auftreten. Enthält ein Prädikat im Template ein RenameSet, trägt diese nicht zur Auflösung der VarSets bei. Alle Attribute, die nicht bereits einem VarSet des Prädikats zugeordnet werden, sind dann dem RenameSet zugeordnet. Die einem RenameSet zugeordneten Attribute werden zu neuen Variablen mit gleichem Namen gebunden und dann später durch die Namenskollisionsauflösung umbenannt. Dies stellt eine Möglichkeit dar, die Regel, dass jedem Attribut nur ein VarSet zugeordnet wird, aufzuweichen. Als Beispiel kann wieder das Prädikat `p` mit Attribut `?a` verwendet werden. Das Template

```
q(?x, ?y) :- p(?x), p(?y)
```

führt zur Regel in Nemo:

```
q(?a, ?a) :- p(?a), q(?a) .
```

Das Template mit RenameSet

```
q(?x, ??*y) :- p(?x), p(??*y)
```

führt hingegen zu dieser Regel in Nemo:

```
q(?a, ?a_1) :- p(?a), q(?a_1) .
```

2.2.2 Prädikat-Typen

In der bisherigen Einführung zur Nemo Template Language wurde gezeigt, wie mit Prädikaten unterschiedlichster Formen in einem einzigen Template umgegangen werden kann. Ein Regeltemplate wie `a(?x) :- p(?x, ??other)` setzt allerdings eine gewisse Struktur der Prädikate, im Beispiel eine Semantik des Attributs an erster Stelle des Prädikats `p`, welche das Prädikat `a` rechtfertigt, voraus. Das Beispiel Template funktioniert für sehr unterschiedliche Strukturen von `p` Prädikaten. Alle diese Prädikate hätten jedoch die Gemeinsamkeit in der Bedeutung des ersten Attributes. Somit funktioniert das Regeltemplate mit einer bestimmten Klasse von Prädikaten, welche einem Typ angehören.

Ein Prädikat-Typ wird hier als eine Abfolge von benannten Keys (Strings) definiert, welche das Präfix bzw. Suffix in der Liste der Attribute eines Prädikats beschreiben. Bei der Regel `a(?x) :- p(?x, ??other)` könnte z.B. die erste Position des Prädikats `p` durch den key „count“ beschrieben werden, wodurch `a` für alle counts aus `p` steht. Somit besteht der Prädikat-Typ darin, dass das Präfix der Liste der Attribute von `p` die einelementige Abfolge ist, welche nur „count“ enthält.

In der Nemo Template Language können solche Prädikat-Typen explizit beschrieben und zur Rust-Compilezeit geprüft werden. Dazu können unterschiedliche Prädikate der Nemo Template Language Instanzen unterschiedlicher Typen in Rust sein. Der jeweilige Rust-Typ modelliert dabei den Prädikat-Typ. Alle diese Typen implementieren den `TypedPredicate` Trait. Dieser beinhaltet die Funktionalität, welche nötig ist, damit ein Prädikat in einem Regeltemplate verwendet werden kann. Der `Basic` Typ, welcher keine besonderen Keys beinhaltet, ist bereits in der Nemo Template Language definiert.

Ein Rust Typ, welche einen Prädikat-Typ modelliert, kann, wie in folgendem Beispiel gezeigt wird, durch Verwendung des `nemo_predicate_type!` Makros in einer einzigen Zeile erstellt werden:

```
nemo_predicate_type!(MyABCType = a b ... c);
```

Im Beispiel wird eine `struct MyABCType` erstellt und für diese der `TypedPredicate` Trait implementiert. Das Makro expandiert zu 99 Zeilen Rust-Code. Der `MyABCType` beschreibt, dass die ersten beiden Attribute den Key mit dem Namen „a“ bzw. „b“ haben und das letzte Attribut durch den Key „c“ beschrieben werden kann. Die Implementierung der `MyABCType struct` beinhaltet die Methoden `a()`, `b()` bzw. `c()`, welche durch ein entsprechendes Regeltemplate aufgerufen werden. Der Compiler stellt sicher, dass die Methoden auch existieren. So wird spätestens zur Compilezeit, mit entsprechendem IDE Support jedoch bereits während des Schreibens des Regeltemplates, die Korrektheit des Prädikattyps geprüft. In einer Nemo Template Language Regel kann der Key eines Typs durch `@`-Syntax angegeben werden:

```
q(?y) :- p(@a: ?x, @b: ?y; ??vars; @c: ?z)
```

Das Beispiel extrahiert alle Werte für ein Attribut mit dem „b“ Key aus dem Prädikat `p`, welches vom Typ `MyABCType` ist. Es ist zu beachten, dass das Präfix bzw. Suffix, welches durch `@`-Keys annotiert ist mittels `;` von den restlichen Variablen zu trennen ist.

Ein weiterer Vorteil, außer das Prüfen zur Compilezeit, entsteht durch die `@?` Syntax. Diese erlaubt eine Variable zu ignorieren, wenn der Typ des Prädikats den entsprechenden Key nicht unterstützt. Dies macht ein Regeltemplate noch flexibler und ermöglicht es in Rust, Funktionen mit generischen Parametern zu implementieren. Folgendes ist ein Beispiel für eine so generische Regel:

```
q(?y) :- p(@a: ?x, @b: ?y; ??vars; @?c: ?x)
```

Diese Regel extrahiert den „b“ Wert. Nur falls `p` den Typ `MyABCType` hat, wird zusätzlich durch das Binden von `?x` an die letzte Position, getestet, dass der erste und letzte Parameter den gleichen Wert haben. Damit unterstützt die Regel zusätzlich zum `MyABCType` auch den `MYABType`, welcher im Beispiel wie folgt, ohne den „c“-Key, definiert ist:

```
nemo_predicate_type!(MyABType = a b ...);
```

2.2.3 Interne Nemo-Repräsentation

Damit die Nemo Template Language funktioniert, wird eine interne Repräsentation von Nemo Regeln benötigt. Die Nemo Bibliothek, welche für die Tests ohnehin eine Abhängigkeit des Projekts ist, bietet ein vollständiges Modell für Nemo Regeln. Eine direkte Verwendbarkeit wurde untersucht und nicht als sinnvoll erachtet. Gründe dafür sind die Prädikate mit Attributen, das Tracken von Eigenschaften, welches zusätzliche Informationen benötigt und das grundlegende Konzept, Regeln in den Prädikaten des Regelkopfes zu speichern. Dies ermöglicht es in der Nemo Template Language nicht für jede Regel zusätzlich eine Collection von Regeln anzugeben, in die sie aufgenommen werden soll. Prädikate die später in bestimmten Fällen zu ignorieren sind, können mit Regeln erstellt werden, ohne das zugehörige Regeln später Teil des Nemo-Programms werden. Außerdem ist die Nemo eigene Repräsentation in einigen Punkten komplexer als für die SPARQL Übersetzung nötig. Insbesondere werden Filter-Conditions teilweise, und alle Sorten von Konstanten, im verwendeten Modell durch Strings beschrieben.

Folgende Grafik zeigt die wesentlichen Bestandteile des im Rahmen dieser Arbeit angefertigten Nemo Modells. Die Pfeile sind durch die Property beschriftet, welche zur Verknüpfung der Modellteile führt:

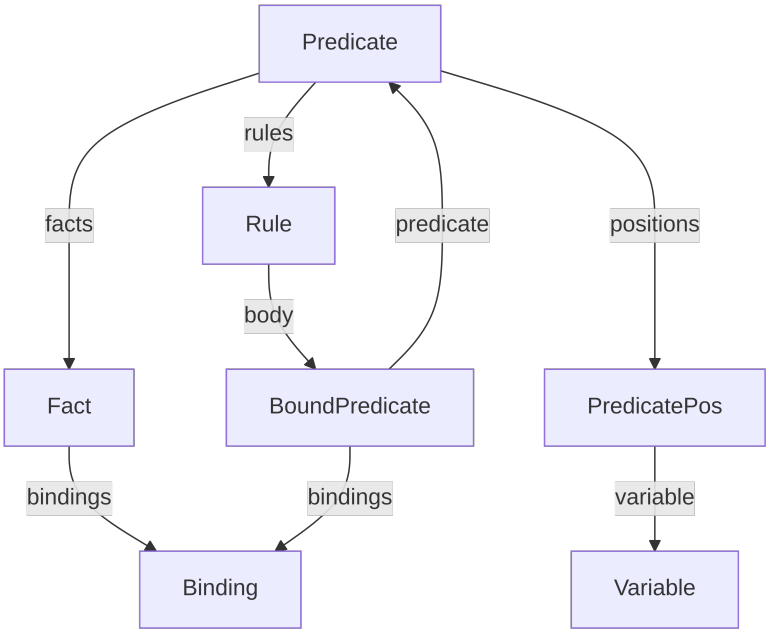

Abbildung 2.3: Übersicht der internen Nemo-Repräsentation

Es ist zu sehen, dass das Prädikat die zentrale Rolle in der Struktur bildet. Ein Prädikat enthält Regeln und Fakten, die es beschreiben. Außerdem werden dessen Positionen durch `PredicatePos` Instanzen repräsentiert, welche Eigenschaften tracken und ein Attribut, welches als „Variable“ im Modell bezeichnet wird, referenzieren. Im Abschnitt 3.3 wird näher auf den Begriff „Variable“ eingegangen. Eine Regel hat einen Regel-Kopf, welcher in der Grafik aus Gründen der Übersichtlichkeit nicht aufgenommen wurde und ähnlich zum Regel-Körper ist. Ein Prädikat im Regel-Körper wird durch eine `BoundPredicate` Instanz abgebildet, welche zu einem `Predicate` die Bindings in der Regel speichert. Ein `BoundPredicate` könnte beispielsweise die Information enthalten, dass die erste Position des Prädikats in der Regel an die Variable `?a` gebunden wird, dabei wäre die Variable `?a` ein `Binding`. Außer Prädikaten können im Regel Body auch Filter Conditions verwendet werden, die in der Grafik nicht dargestellt sind. Ein `Fact` unterstützt in Nemo nur Konstanten als Bindings.

Für `Variable` und `Predicate` wurde der `Eq` und `Hash` Trait implementiert, sodass Instanzen aufgrund ihrer Speicheradresse verglichen werden. Dies ermöglicht es, Variablen und Prädikate mit gleichem Namen zu unterstützen, welche dann im Rahmen der Namenskollisionsauflösung (Unterabschnitt 2.2.5) durch Anfügen eines Suffixes eindeutig gemacht werden. Dies ist notwendig, z.B. wenn ein Prädikat in einer Unterfunktion erstellt wird, welche mehrfach aufgerufen wird, was zu unterschiedlichen Prädikaten mit gleichem Namen führt. Außerdem werden `Variable` und `Predicate` nur sehr selten direkt verwendet. Meistens werden die jeweiligen Smart-Pointer-Wrapper `VarPtr` und `PredicatePtr`, welche in Rust gecloned werden können ohne die Speicheradresse und damit, aufgrund der `Eq` Implementierung, die Identität zu verlieren.

Unterschiedliche Nemo Regel Features werden durch die `Binding` `enum` unterstützt:

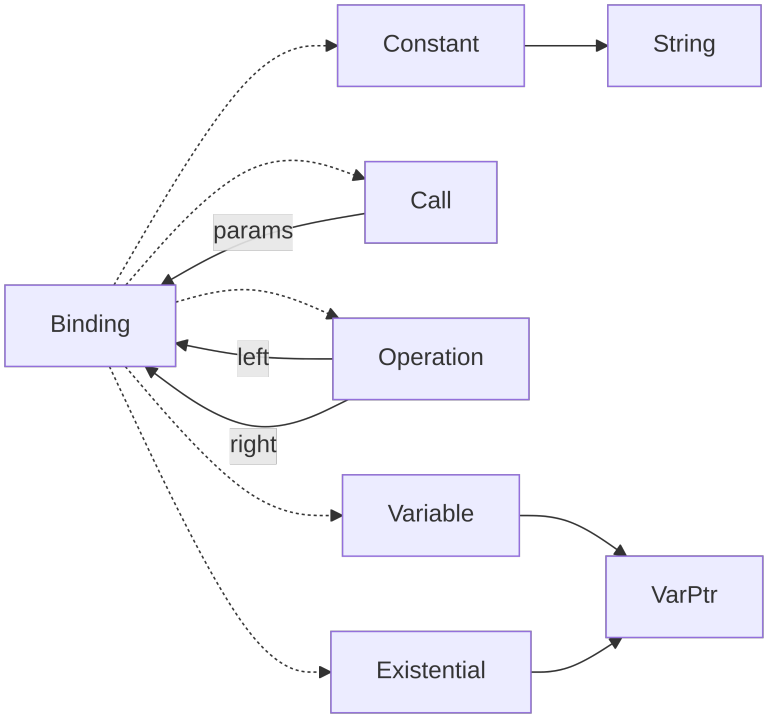

Abbildung 2.4: Übersicht der Binding-Enumeration für Positionen in Prädikaten

Die `enum` Varianten `Call` und `Operation` sind in einer gleichnamigen `struct` implementiert, welche von der `enum` Variante nur referenziert wird. Diese Information ist aus Gründen der Übersichtlichkeit nicht in der Grafik abzulesen. Zahlreiche Typen implementieren den `From` Trait in Rust, um zu `Binding` konvertierbar zu sein, so z.B. unterschiedliche Builtin-Typen in Rust wie Zahlen, Strings und Boolean-Werte. Die Konstanten werden nach RDF Syntax als String gespeichert. Dies kann von der im Projekt verwendeten spargebra Rust-Bibliothek generiert und von Nemo interpretiert werden. Somit bildet es eine einfache und flexible Schnittstelle.

Zusätzlich zur Modellierung von Nemo Regeln werden auch Features der Nemo Template Language modelliert. Dazu gibt es die `ProtoPredicate` `enum` für Dinge, welche zu Prädikaten und Filtern in einem Regel-Körper führen und `ProtoBinding` `enum` für Dinge, die zu `Bindings` in einem Prädikat werden können. Konkret ist z.B. ein VarSet ein `ProtoBinding`, welches dann zu einzelnen Variablen in Nemo aufgelöst wird. `ProtoPredicate` und `ProtoBinding` sind insbesondere bei der Implementierung der Nemo Template Language Makros (Unterabschnitt 2.2.4) relevant.

2.2.4 Nemo Template Language Makros

Das Zentrum der Nemo Template Language bilden zwei Makros, welche es erlauben, ein Nemo Regel Template im Rust-Code zu verwenden. Wie in Unterabschnitt 2.2.3 beschrieben werden Regeln im Prädikat, welches den *Regel-Kopf* ↗ bildet, gespeichert. Das `nemo_def!` Makro erlaubt es, ein Nemo Regel Template zu schreiben, wobei die resultierende Regel in einem neu definierten Prädikat gespeichert wird. Dem gegenüber steht das `nemo_add!` Makro, welches auch erlaubt, ein Nemo Regel Template zu schreiben, jedoch wird die

resultierende Regel einem existierenden Prädikat hinzugefügt. Bei der Erstellung eines neuen Prädikats müssen die Attribute für dieses auf Basis der Regel abgeleitet werden, dadurch verhält sich das `nemo_def!` Makro etwas anders. Insbesondere bei der Zuordnung der Attribute zu VarSets (Unterabschnitt 2.2.1) können nicht die Attribute des Prädikates im *Regel-Kopf* ↗ zur Bestimmung der VarSets, sondern werden die VarSets zur Bestimmung der Attribute des Prädikates im *Regel-Kopf* ↗ verwendet. Folgendes ist eine Regel, die für `nemo_add!` aber nicht für `nemo_def!` valide wäre:

```
// sowohl p als auch q sind Rust-Variablen im aktuellen Scope
nemo_add!(p(??a) :- q(??a, ??b));
```

Würde dieses Regeltemplate mit dem `nemo_def!` Makro verwendet, wäre die Zuordnung der Attribute zu den VarSets `??a` oder `??b` nicht wohldefiniert, da beide sich nur durch den *Regel-Kopf* ↗ unterscheiden.

Ein weiterer Unterschied der beiden Makros besteht darin, dass bei `nemo_def!` nach der Regel der Typ (Prädikat Typen in Unterabschnitt 2.2.2) separiert durch `;` angegeben werden muss und optional der Name des Prädikats angegeben werden kann. Der Name der Rust Variable, in der das neue Prädikat mit der Regel gespeichert wird, ist durch den Namen des Prädikats, der im Template verwendet wird, gegeben. Wenn nicht ein anderer explizit angegeben wurde, wird dieser auch für das Prädikat später beim Erstellen des Nemo-Programms verwendet. Ein expliziter Name ist sinnvoll, wenn das Template in einer in mehreren Situationen verwendeten Funktion verwendet wird, das Prädikat aber zur besseren Verständlichkeit einen der konkreten Situation angepassten Namen haben soll. Gibt es mehrere Prädikate mit gleichen Namen, werden diese im Rahmen der Namenskollisionsvermeidung (siehe Unterabschnitt 2.2.5) durch ein Suffix umbenannt. Im Rahmen dieser Arbeit wird zur Beschreibung einer Nemo Regel der im Regel Template verwendete Prädikat Name verwendet, ohne gesondert darauf einzugehen, dass der tatsächliche Name im späteren Nemo-Programm sich davon unterscheiden könnte.

Trotz der genannten Unterschiede sind beide Makros sehr ähnlich. Ein `nemo_def!` Makroaufruf kann durch manuelles Erstellen des Prädikats und nachfolgendem `nemo_add!` Makroaufruf ersetzt werden. Dies wird in komplexeren Fällen auch nötig, da nicht immer die Attribute des neuen Prädikats aus einer einzelnen Regel abgeleitet werden können. In solchen Fällen muss das Prädikat manuell mit den entsprechenden Attributen erstellt werden. Beide Makros unterstützen neben Regeln auch einfache Fakten.

Alle Makros der Nemo Template Language sind als deklarative Makros in Rust implementiert. Hier ein Beispiel, wie ein `nemo_def!` Makro expandiert wird:

```
nemo_def!(new_predicate(?x) :- existing_predicate(?x); Basic);
```

Expandierter Rust-Code:

```
let mut builder = crate::nemo_model::RuleBuilder::new();
builder.set_predicate_name("new_predicate");
builder.add_head_binding(
    crate::nemo_model::ProtoBinding::NamedConnection("x".to_string())
);
builder.start_body_atom();
builder.add_body_binding(
    crate::nemo_model::ProtoBinding::NamedConnection("x".to_string())
);
builder.finalize_atom({ &existing_predicate }.get_predicate(), false);
let new_predicate = builder.to_predicate::<Basic>();
```

Rust erkennt dabei korrekt, dass der `new_predicate` Identifier aus dem ursprünglichen Makroaufruf kommt, `builder` jedoch nicht. Deswegen kann `new_predicate` in späterem Rust-Code verwendet werden, es kommt aber zu keinen Kollsionen, wenn mehrere Makros die Variable `builder` verwenden. Es ist erkennbar, dass eine einzelne Variable mit einfachem `?` Präfix als `NamedConnection` bezeichnet wird, da sie eine einfache Verbindung anhand eines Namens zwischen Attributen unterschiedlicher Prädikate herstellt. `existing_predicate` hat in Rust einen Typ, der den `TypedPredicate` Trait implementiert, welcher das zugrundeliegende Prädikat über die `get_predicate()` Methode zur Verfügung stellt. `new_predicate` hat den Typ `Basic`, welcher ebenfalls den `TypedPredicate` Trait implementiert. Der zweite Parameter der `finalize_atom` Methode gibt an, ob das Prädikat mittels stratifizierter Negation negiert werden soll. Die Nemo Template Language unterstützt dies durch ~ wie in Nemo. Die `to_predicate` Funktion führt den Prozess des Auflösens der Nemo Template Language Features zu Nemo Features durch. Beispielsweise werden die Prädikat-Parameter Bindings der Nemo Template Language (`ProtoBinding`) zu validen Bindings in Nemo (`Binding`) aufgelöst. Dies schließt insbesondere das Auflösen der VarSets ein. Die äquivalente Funktion für `nemo_add!` ist `perform_add()`.

Die verwendete IDE RustRover unterstützt Autovervollständigung, Syntaxhighlighting und Linting auch innerhalb von Makroaufrufen. Allerdings wird zum Zeitpunkt der Implementierung dies nicht für verschachtelte Makros unterstützt, außerdem ist die Möglichkeit der Abstraktion in Rust für Makros nicht umfangreich unterstützt. Deswegen wurden die `nemo_add!` und `nemo_def!` Makros in einer schwer wartbaren fraktalen Struktur geschrieben, wobei gleiche Pattern an allen Stellen, wo diese auftreten geinlined wurden. Dadurch umfassen beide Makro-Definitionen zusammen über 750 Zeilen. Die Namen für die Makrovariablen bilden dabei eine hierarchische Struktur. Eine der längsten Variablennamen ist beispielsweise `head_aggregate_front_connection_name`. Dies bezeichnet den connection_name, wenn dieser im Head der Regel innerhalb einer Aggregationsexpression im vorderen Teil des Prädikats, also vor den VarSets, steht. Die Sprache zur Definition deklarativer Makros in Rust schränkt die Möglichkeiten ein, welche Symbole in welcher Situation verwendet werden können, um ein effizientes Parsen zu garantieren. So müssen Aggregationsfunktionen in der Nemo Template Language mit Präfix `%` statt wie in Nemo mit Präfix # aufgerufen werden, da das # Symbol in Rust auch in Expressions zum Aufruf von Makros verwendet werden kann.

Neben den beiden Hauptmakros zur Regelerstellung gibt es noch zahlreiche weitere Makros, welche Teile von Regeln innerhalb von Rust-Code erstellen können:

Makro	Beschreibung und Beispiel
`nemo_declare!`	erstellt ein Prädikat ohne eine dazugehörige Regel mit zuvor explizit erstellten Attributen. Beispiel: `nemo_declare!(p(a, b));`
`nemo_var!`	erstellt eine neue Nemo-Variable mit gegebenem Namen. Es werden auch existentielle Variablen unterstützt. Beispiel: `nemo_var!(x)`, `nemo_var!(!x);`
`nemo_iri!`	erstellt eine Nemo IRI, optional mit einem Präfix, das zuvor in einer Rust-Variable definiert wurde. Beispiel: `nemo_iri(rdf => type)`

Makro	Beschreibung und Beispiel
`nemo_call!`	erstellt einen Nemo-Funktionsaufruf, bei dem alle Parameter explizit definierte Variablen oder genestete Rust-Expressions sein müssen. Beispiel: `nemo_call!(STR; s)`

`nemo_filter!`	erstellt eine Nemo-Filter-Kondition basierend auf Strings. Implizite Variablen wie `?1` werden unterstützt.
	Beispiel:
	`nemo_filter!(`
	`  "fullStr(", ?1, ") != fullStr(", ?r, ")"`
	`)`

`nemo_condition!`	wie `nemo_filter!`, aber nur für drei Parameter (linke Seite, Operation, rechte Seite). Beispiel: `nemo_condition!(?1, " != ", ?r)`

`nemo_predicate_type!`	erstellt eine neue Rust-Struct, die den `TypedPredicate`-Trait implementiert.
	Beispiel:
	`nemo_predicate_type!(`
	`  FirstParameterIsCountPredicate = count ...`
	`);`

Zwei Makros können verwendet werden, um mehrere Dinge mittels Rust-Code in einer Rust-Variable zu aggregieren und dann direkt eine Menge von Dingen auf einmal in einem Regeltemplate zu verwenden. Konkret sind dies das `nemo_atoms!` Makro, welches logische Atome, wie ein Prädikat mit gebunden Parametern oder einer Filter-Kondition aggregieren kann, und das `nemo_terms!` Makro, welches logische Terme, wie Variablen oder Funktionsaufrufe, aggregieren kann. Das `nemo_atoms!` Makro erstellt eine spezielle `Multi` Variante der `ProtoPredicate` Enumeration und das `nemo_terms!` Makro eine `TermSet` Instanz. Ein `TermSet` kann nicht nur in Regeln verwendet werden, sondern bietet auch Funktionalität zum Generieren von Attributen für die manuelle Erstellung eines Prädikats und zum Deduplizieren unter Beibehaltung der Reihenfolge. Das `nemo_terms!` Makro unterstützt zusätzlich einen Mappingsyntax wie `nemo_terms![term_list => transform_func]` wobei die Elemente eines iterierbares Objekts `term_list` nicht direkt, sondern nach Aufruf der `transform_func` Funktion zum `TermSet` zusammengefasst werden. Beide Makros aggregieren Dinge, in dem sie jeweils ihren eigenen Ergebnistyp enthalten können. So kann z.B. ein Rust-Loop, welcher Nemo Atome aggregiert, aussehen:

```
let mut atoms = nemo_atoms![];
for element in some_list {
    atoms = nemo_atoms![atoms, element.get_atom()]
}
```

2.2.5 Interne Nemo-Repräsentation zu Nemo-Programm

Die Eingabe für das Generieren eines Nemo-Programms bildet ein Nemo Prädikat. Für dieses Prädikat werden die Regeln, von denen es abhängt, generiert. Rekursiv werden auch die Programme, für die Prädikate die in den Regeln vorkommen depth-first generiert. Dabei sind einige Dinge zu beachten:

- Für Regeln mit leerem *Regel-Körper*↗ muss ein `dummy()` Prädikat generiert werden und in den *Regel-Körper*↗ aufgenommen werden, da in Nemo ein *Regel-Körper*↗ nicht leer sein darf. Eine leere Regel ist dabei unterschiedlich von einem Fakt, da in Nemo für Fakten andere Features unterstützt werden. Existentiell quantifizierte Variablen können in der verwendeten Nemo Version nicht in Fakten verwendet werden. Außerdem unterstützen Fakten keine Berechnungen, wie z.B. Funktionsaufrufe. Zwar ergibt jede Berechnung in einem Fakt eine Konstante, die prinzipiell auch im Vorhinein berechnet werden kann, allerdings ist eine Berechnung direkt in Nemo zu bevorzugen, da dies zu einer einheitlichen Implementierung der Berechnungen führt und den Implementierungsaufwand, die Berechnung während der Übersetzung zu unterstützen, erspart.
- Nemo unterstützt in der verwendeten Version keine Prädikate mit Arität Null. In der Praxis kommet es jedoch bei der Übersetzung vor, dass ein Regeltemplate zur Arität Null führt und dies auch das semantisch korrekte Ergebnis liefert. Solche Prädikate mit Arität Null werden zum Zeitpunkt der Transformation mit einer Dummyposition versehen, welche immer an die Konstante `arity_zero` gebunden wird. Damit an allen Stellen mit Arität Null gleich umgegangen wird, führt ausschließlich die Funktion `make_non_empty()` diese Ersetzung durch.
- Für unterschiedliche Prädikate mit gleichem Namen ist eine Namenskollisionsauflösung durchzuführen werden. Dazu werden alle bisher verwendeten Namen gespeichert und im Kollisionsfall ein numerisches Suffix angefügt. Dadurch ergeben sich Namen wie `pred`, `pred_1`, `pred_2` für drei unterschiedliche Prädikate mit Namen `pred`. Unterschiedliche Variablen mit gleichen Namen müssen innerhalb einer Regel umbenannt werden. Dafür wird dieselbe Funktionalität wie für Prädikate verwendet. Der Zustand ist dabei lokal innerhalb einer Regle in einem `VariableTranslator` Objekt gespeichert.
- Es gibt Differenzen zwischen Nemo und SPARQL, welche Variablen-Namen gültig sind, z.B. erlaubt SPARQL eine Ziffer in erster Position des Variablennamens Nemo nicht. Insbesondere bei automatischen Variablen-Namen bei SPARQL-Transformationen in der spargebra Rust-Bibliothek tritt dies häufig auf, da diese zum Teil auf Basis von UUIDs oder BNode Namen generiert werden. Ein `VariableTranslator` Objekt löst dies, indem alle nicht alphanumerischen Symbole aus dem Variablennamen entfernt werden und falls die Variable nicht mit einem ASCII Buchstaben beginnt, das Präfix `var_` angefügt wird. Dies ist ausreichend für die durchgeführten Tests, trifft jedoch nicht exakt die validen Variablennamen in Nemo. Beispielsweise ist der Umlaut „ö“ ein alphanumerisches Zeichen, das zwar von SPARQL, aber nicht von Nemo unterstützt wird.

2.3 SPARQL Ergebnistypen in Datalog

Das Design der SPARQL zu Nemo Übersetzung erfordert eine Festlegung, wie SPARQL Ergebnisse dennoch möglich werden, obwohl sie, z.B. durch Beachtung der Reihenfolge, nicht *direkt* durch ein Nemo Prädikat *repräsentiert*↗ werden können. Ein Nemo Prädikate *repräsentiert*↗ eine Menge von Tupel. Somit lassen sich ungeordnete Ergebnisse einer

SPARQL-Query ohne Duplikaten direkt darstellen. Andere Typen lassen sich durch zusätzliche Prädikatpositionen mit spezieller Bedeutung darstellen. Hier die 4 Ergebnistypen:

Prädikat-Typ-Definition	Beschreibung
`SolutionSet = ...`	Menge von Tupel: keine besonderen Positionen
`SolutionMultiSet = count ...`	Menge mit Duplikaten: erste Position gibt Multiplizität an
`SolutionSequence = index ...`	Geordnete Liste: erste Position gib Index an
`SolutionExpression = ... result`	Berechnung: mögliche Eingabekombination gefolgt von Ergebnis.

Eine so definierte geordnete Liste kann auch Multiplizität darstellen, in dem das gleiche Ergebnis an mehreren Indices auftritt. Weitere Details zu `SolutionExpression` werden im Abschnitt 3.4 vorgestellt. Eine SPARQL-Query ergibt grundsätzlich eine `SolutionSequence`. Eine Ausnahme dabei bildet ein Ergebnis vom Typ `SolutionSet`, wenn das `DISTINCT` Keyword verwendet wurde (siehe Unterabschnitt 3.7.1). Der `SolutionMultiSet` Typ wird ausschließlich für Zwischenergebnisse verwendet und bietet für viele SPARQL Operationen eine einfachere Implementierung, da sie ohne Reihenfolge definiert sind und Berechnungen mit Reihenfolge in Datalog häufig weniger effizient und komplexer zu implementieren sind.

2.4 Überblick der Übersetzung

Die Evaluierung einer SPARQL-Query kann grundsätzlich schematisch wie in Abbildung 2.5 dargestellt werden. Es werden nur SPARQL-Queries, welche auf einem Triple Graph angewendet werden betrachtet, Queries über mehrere Graphen oder Update-Queries werden in dieser Arbeit nicht berücksichtigt.

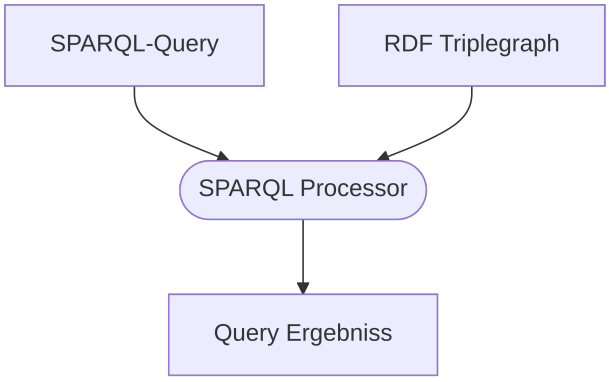

Abbildung 2.5: High-Level Queryevaluierungsablauf

Soll Nemo als SPARQL Processor verwendet werden, wird zusätzlich eine Übersetzung von SPARQL zu Nemo benötigt. Eine solche Übersetzung wurde im Rahmen dieser Arbeit entwickelt. Die Evaluierung einer SPARQL Query mit Nemo wird in Abbildung 2.6 dargestellt.

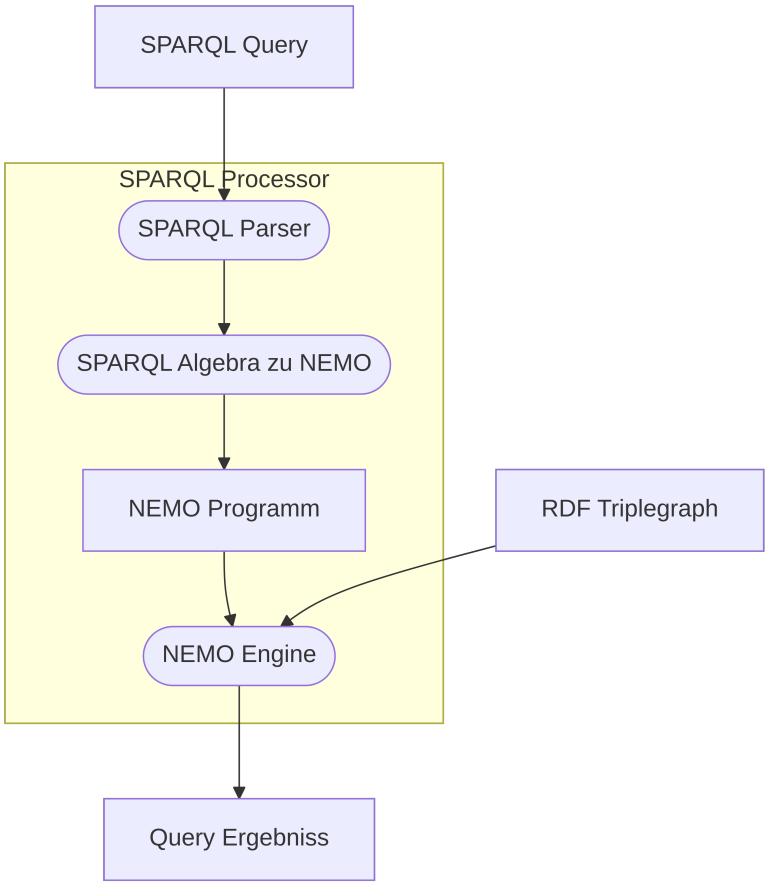

Abbildung 2.6: SPARQL2Nemo Queryevaluierungsablauf

Die Übersetzung ist grundsätzlich eine String-Transformation. Als Eingabe wird ein SPARQL-Query-String entgegengenommen und die Ausgabe ist ein Nemo-Programmfragment. Dieses verwendet ein definiertes dreistelliges Prädikat `input_graph(s, p, o)` welches den *Triple-Graph*↗ auf dem die SPARQL-Query evaluiert werden soll *repräsentiert*↗.

Die Übersetzung eines SPARQL-Query-Strings zu einem Nemo String wird von der `translate()` Funktion durchführt. Diese sei hier vollständig abgebildet:

```
pub fn translate(query_str: &str) -> Result<String, TranslateError> {
    let query = Query::parse(query_str, None)?;
    let input_graph = PredicatePtr::new(
        "input_graph",
        vec![VarPtr::new("s"), VarPtr::new("p"), VarPtr::new("o")]
    );
    let mut translator = QueryTranslator::new(input_graph);
    let solution_predicate = translator.translate_query(&query)?;
    Ok(construct_program(solution_predicate.as_ref()))
}
```

Außer einem Nemo-String kann die Funktion auch einen Fehler erzeugen, welcher entweder durch einen Fehler beim Parsen der Query oder durch einen Fehler beim Übersetzen zustande kommt. Der Übersetzungsprozess erfolgt in vier Schritten:

1. **Parsing der SPARQL-Query:** Dafür wird die Rust-Bibliothek spargebra[1] verwendet, welche die Parsing Komponente von Oxigraph[2], einer Graphdatenbankimplementierung in Rust, ist. Spargebra behandelt bereits einige Fehlerfälle und führt im SPARQL Standard definierte Transformationen durch. Dies erleichtert weitere Schritte.
2. **Erstellen des `QueryTranslator`:** Der `QueryTranslator` wurde im Rahmen dieser Arbeit entwickelt und ist für die Übersetzung verantwortlich. Der `QueryTranslator` wird mit dem Prädikat, welches den Eingabe-Graphen *repräsentiert*↗, konfiguriert. Außerdem wird im Konstruktor ein Prädikat für null Values (siehe Abschnitt 3.1) und ein `VariableTranslator` (siehe Abschnitt 3.3) erstellt.
3. **Generieren der internen Nemo Repräsentation (Unterabschnitt 2.2.3) aus der SPARQL-Algebra:** Die `translate_query()` Funktion gibt ein Prädikat, welches das SPARQL *Query-Ergebnis*↗ *repräsentiert*↗, zurück. Da diese Übersetzung ein wesentlicher Inhalt dieser Arbeit ist, folgt ein näherer Überblick im Rest des Kapitels.
4. **Erstellen eines Nemo-Programm-Strings aus der internen Nemo Repräsentation:** Dies folgt dem bereits in Unterabschnitt 2.2.5 vorgestellten Vorgehen.

Um den SPARQL-Algebra Baum in die interne Nemo-Repräsentation zu übersetzen, werden die Knoten rekursiv bottom up übersetzt. Es wird für jeden Knoten der Algebra ein Prädikat in Nemo generiert, welches das Ergebnis der Evaluation des Teilbaums *repräsentiert*↗. Bei der Übersetzung einer Algebra Operation werden ausschließlich neue Prädikate im *Regel-Kopf*↗ und Prädikate aus der Übersetzung des aktuellen Teilbaums, einschließlich der neuen Prädikate, im *Regel-Körper*↗ verwendet. Durch die Baumstruktur der SPARQL-Algebra wird direkt ein kanonischer azyklischer Abhängigkeitsgraph der Prädikate des Nemo-Programms und damit eine Stratifizierung (siehe Unterabschnitt 1.5.2) impliziert. Ein nicht stratifiziertes Programm kann nur auftreten, wenn

- eine Implementierung eines einzelnen SPARQL-Algebra Knotens ein nicht stratifiziertes Programmfragment liefert;
- das `input_graph` Prädikat vom SPARQL Ergebnis Prädikat abhängt;
- von der strikten Struktur, z.B. durch Optimierungen, abgewichen wird.

2.5 Testsuite

Die Testsuite verwendet die in Rust verfügbaren Test-Features, insbesondere das `#[test]` Attribut. Fast alle Testcases sind SPARQL-Queries und Eingabedaten mit erwarteten Ergebnissen. Die SPARQL-Query wird zunächst übersetzt und dann mittels Nemo auf den Eingabedaten ausgeführt. Schließlich wird das Ergebnis mit dem erwarteten Ergebnis verglichen. Nemo wird in den Tests direkt importiert und verwendet, wodurch Nemo eine Abhängigkeit des Rust-Projekts wird. Es gibt grundsätzlich Tests für alle Übersetzungsfunktionen. Die Tests wurden während der Implementierung der Übersetzungsfunktionen mit entwickelt, was ein intensiveres Testen, von während der Implementierung als komplex empfundenen Programmteilen, ermöglicht. Die Tests tragen wesentlich zur Korrektheit der

[1] https://docs.rs/spargebra/latest/spargebra/
[2] https://github.com/oxigraph/oxigraph

Übersetzung bei. Es gibt eine gesonderte Evaluation, da die Testsuite nicht unabhängig ist, das heißt, fehlendes Verständnis für ein SPARQL-Feature kann sowohl zu einer falschen Implementierung als auch zu fehlenden dazu passenden Tests führen. Außerdem wurde die Implementierung kontinuierlich angepasst, sodass die Testsuite keine Fehler erkennt. Eine fehlende Generalisierung der Korrektheit auf reale Suchanfragen kann jedoch ohne separate Evaluation nicht beurteilt werden. Features, bei denen Limitierungen in Nemo oder der Implementierung bekannt sind, und daher im Test nicht erfolgreich sind, werden in der Testsuite nicht getestet.

Für die Testimplementierungen werden die Funktionen `assert_sparql()` und `assert_sparql_multi()` verwendet. Beide nehmen als Eingabe die SPARQL-Query als String, die Inputs als Nemo-Programm-Fragment, welches das Prädikat `input_graph(?s, ?p, ?o)` beinhaltet, und das erwartete Ergebnis. Das erwartete Ergebnis wird als String übergeben und hat ein speziell definiertes Format. Dabei können Tupel durch „;“ oder eine neue Zeile getrennt werden und Werte innerhalb eines Tupel sind durch „,“ separiert. Beginnt und endet der String mit „[“ bzw. „]“ wird jedem Tupel dessen Index als erster Wert hinzugefügt (siehe Darstellung von Ergebnissen mit Reihenfolge in Abschnitt 2.3). Die Reihenfolge der Tupel ist grundsätzlich nicht relevant da die eigentliche Reihenfolge, wenn relevant, durch den Index in jedem Tupel gegeben ist. Die Funktion `assert_sparql_multi()` ignoriert die erste Tupleposition und kann damit verwendet werden, falls die erste Position der Index ist, aber die Reihenfolge egal ist. Wie der Name bereits ausdrückt, wird die Multiplizität jedes Ergebnisses beachtet. Beide Funktionen liefern im Fehlerfall eine möglichst gut formatierte Fehlermeldung, welche fehlende und extra Tupel im Queryergebnis anzeigt.

Die Namen der Testfälle beginnen mit dem Feature-Namen der Übersetzungsfunktion. Beispielsweise gibt es für die Übersetzungsfunktionen für Basic-Graph-Patterns, `translate_bgp()` und `translate_bgp_multi()` die Testfunktionen `bgp_simple()`, `bgp()` und `bgp_multi()`. Jede Testfunktion kann mehrere SPARQL-Queries testen und in jeder SPARQL-Query können unter Umständen mehrere Tests stattfinden, z.B. kann in mehreren `BIND` Zeilen einer Query jeweils ein anderer Test stattfinden. Die Testsuite umfasst 94 Aufrufe der Funktionen `assert_sparql()` und `assert_sparql_multi()` in 89 Testfällen und damit 94 zu testende SPARQL-Quieries. Alle Tests laufen in der finalen Implementierung mit der verwendeten Nemoversion erfolgreich. Da Nemo globale Locks für das Timing verwendet, lassen sich die Tests nicht parallelisieren und müssen mit der Umgebungsvariable `RUST_TEST_THREADS=1` ausgeführt werden. Dennoch wurden die Tests möglichst effizient gehalten und laufen in ca. 16,7 s vollständig durch. Auch die Korrektheit des Codes, welcher die als String gegebene Erwartung mit dem tatsächlichen Ergebnis vergleicht, wird getestet.

3 Implementierung der SPARQL Features

In diesem Kapitel werden die einzelnen SPARQL Features nacheinander vorgestellt und beschrieben, wie diese in Nemo-Programme (Unterabschnitt 1.5.2) übersetzt werden können.

3.1 Umgang mit Unbound-Werten

In SPARQL ist ein einzelnes Ergebnis einer Query ein „solution mapping“ (eine partielle Funktion) $\mu : V \rightarrow$ RDF-T. Dabei ist V die Menge der SPARQL Variablen und RDF-T die Menge der möglichen Werte in RDF (Terms). In Unterabschnitt 1.5.1 wurde bereits vorgestellt, wie diese Darstellung zu einer Darstellung eines Ergebnisses mithilfe eines Tupel und einem Prädikat mit Attributen korrespondiert. Die Domänen ($dom(\mu)$) für unterschiedliche Ergebnisse μ eine SPARQL-Query können sich unterscheiden. Das *Ergebnis*↗ eines Prädikats in Nemo enthält jedoch nur Tupel gleicher Länge.

Eine Variante unterschiedliche Domains abzubilden wäre ein eigenes Prädikat für jede mögliche Domäne. Dies kann jedoch zu sehr großen Nemo Programmen führen, da die Zahl der möglichen Domains exponentiell von der Anzahl der Variablen in der Query abhängt. Eine bessere Alternative ist die Einführung eines speziellen Wertes, genannt „*UNDEF-Marker*↗ “. Der Name ergibt sich aus dem `UNDEF` Schlüsselwort beim `VALUES` Feature in SPARQL, welcher das Fehlen eines Wertes darstellt. Dies führt jedoch nicht direkt zu korrektem Verhallten bei vielen SPARQL Features. Anpassungen, wie bei diesen der *UNDEF-Marker*↗ unterstützt werden kann, sind im weiteren Verlauf dieser Arbeit vorgestellt. Der *UNDEF-Marker*↗ verhält sich bei vielen Features eher wie eine Wildcard als ein konkreter Wert. Innerhalb von Expressions (Abschnitt 3.4) führen nicht definierte Variablen direkt zu einem Fehler. Dies hat zur Folge, dass der *UNDEF-Marker*↗ innerhalb von Expressions nicht weiter berücksichtigt werden muss, abgesehen von der `bound( )` Expression. (Unterabschnitt 3.4.8).

Es ist nötig, dass der *UNDEF-Marker*↗ nicht in den Eingabedaten der Query oder (Zwischen-) Ergebnissen vorkommt. Dies kann in Nemo sichergestellt werden, indem der *UNDEF-Marker*↗ ein, durch eine existentielle Regel generierter, BNode ist. In der `QueryTranslator` Klasse gibt es das, im Konstruktor erstellte, `undefined_val` Prädikat, welches die einelementige Menge mit dem *UNDEF-Marker*↗, z.B. `{(_:0)}`, *repräsentiert*↗. Da ein sich ändernder BNode die Verständlichkeit beim Betrachten der Ergebnisse verringert, gibt es bei der Übersetzung auch

die Option, dass der *UNDEF-Marker* ↗ die einfache Nemo IRI `UNDEF` ist. Alle Implementierungen können grundsätzlich mit beliebigen Werten als *UNDEF-Marker* ↗ umgehen.

3.2 Umgang mit Fehlern

In Nemo führen Fehler bei Berechnungen, z.B. in Funktionen, zu fehlenden Tupels im Ergebnis. Dies gibt die zu verwendende Repräsentation von Fehlern durch fehlende Tupels vor, da sonst eine umfangreiche Implementierung zum Abfangen von Fehlern nötig wird. Eine Konsequenz dieser Repräsentation ist die Notwendigkeit von Negation zum Prüfen von Fehlern. Da Negation in Datalog ein nicht triviales Feature ist, könnte dies als Nachteil gesehen werden. Stratifizierte Negation bewirkt beispielsweise, dass durch das Testen auf Fehler das Programm nicht stratifiziert wird, wenn der Eingabe-Graph vom Ergebnis der Query abhängt, was zwar kein Problem darstellt, aber dennoch unerwünscht ist (Abschnitt 1.1). Eine Alternative ohne stratifizierte Negation und dafür mit existentiellen Regeln wird exemplarisch in Unterabschnitt 3.7.6 vorgestellt. Diese funktioniert jedoch nicht in allen Fällen.

Es gibt in der Implementierung keine Unterscheidung unterschiedlicher Fehlertypen.

3.3 Umgang mit Variablen

In dieser Arbeit gibt es in unterschiedlichen Zusammenhängen Variablen, die mit `?` Präfix gekennzeichnet sind:

- SPARQL-Query Variablen (siehe Abschnitt 1.4)
- Nemo Regel Variablen (siehe Unterabschnitt 1.5.2)
- Nemo Template Language Variablen (siehe Unterabschnitt 2.2.1)
- Prädikat-Attribute (siehe Unterabschnitt 1.5.1)

Im Rahmen der Übersetzung kann dieselbe Variable in mehreren dieser Zusammenhänge verwendet werden. Typischerweise verläuft eine Variable folgende Schritte:

Abbildung 3.1: Übergang zwischen verschiedenen Variablen

Eine SPARQL Variable nimmt typischerweise eine Position in einem Basic-Graph-Pattern ein und wird für das entsprechende Prädikat ein Prädikat-Attribut. Dieses Attribut wird dann in weiteren Regeln verwendet, wodurch es nach den Regeln der Nemo Template Language (Abschnitt 2.2) auch ein Attribut der daraus abgeleiteten Prädikate werden kann. Am Schluss werden die Variablen in den Nemo Regeln aus den Prädikat-Attributen abgeleitet. Ein Übergang zwischen Nemo Variable und Variable der Nemo Template Language findet im Prozess der Verallgemeinerung einer Nemo Regel zu einem Regeltemplate statt. Außerdem steht eine Nemo Template Language Variabele in jeder Regelanwendung für ein konkretes Prädikat-Attribut. Eine explizit in Rust erstellte Variable (`VarPtr` in Unterabschnitt 2.2.3) kann auch in einem Regeltemplate verwendet werden und steht dann direkt für sich selbst als Prädikat-Attribut.

SPARQL Variablen sind durch ihren Namen eindeutig. Prädikat-Attribute müssen jedoch keine SPARQL Variablen sein und sind im Programmcode eindeutig basierend auf ihrer Speicheradresse. Für das Mapping von spargebra SPARQL-Variablen zu Nemo Template Language Prädikat-Attributen hat das `QueryTranslator` Objekt eine `VariableTranslator` Instanz[1], welcher Prädikat-Attribute (im Code `VarPtr`) basierend auf ihrem SPARQL Variablen-Namen zurückgibt. So können neue Variablen als Prädikat-Attribute (`VarPtr`) jederzeit frei erstellt werden und mit SPARQL Variablen einheitlich zusammen verwendet werden, ohne dass Kollisionen möglich sind. (Siehe Kollisionsauflösung in Unterabschnitt 2.2.5)

3.4 Expressions

Expressions sind eine Art von Operationen in der SPARQL-Algebra, die bei gegebener Variablenzuweisung zu einem Literal, BNode oder zu einer IRI evaluiert werden können. Eine Expression-Operation wird, wie alle SPARQL Algebra-Operationen, zu einem Nemo Prädikat übersetzt. Für Expressions hat dieses Prädikat den Typ `SolutionExpression`, welcher wie folgt, definiert ist (siehe auch Unterabschnitt 2.2.2):

```
nemo_predicate_type!(SolutionExpression = ... result);
```

Der Prädikattyp `SolutionExpression` umfasst Prädikate, deren letzte Stelle „result“ heißt. Die anderen Stellen werden verwendet, um die möglichen Inputparameterkombinationen zu erfassen. Das heißt für die Expression ?a + ?b ist das resultierende Prädikat dreistellig, wobei z.B. die erste Stelle den Wert für ?a, die zweite Stelle den Wert für ?b und die dritte Stelle das Ergebnis der Berechnung ?a + ?b enthält. Da eine Expression immer auf einer konkreten Menge von Solution Mappings (Unterabschnitt 1.5.1) evaluiert wird, müssen nicht alle theoretisch denkbaren Kombinationen der Inputparameter betrachtet werden, sondern nur solche, die in der konkreten Menge von Solution Mappings vorkommen. Dadurch ist die Größe der Menge, die das Prädikat der Expression *repräsentiert*↗ kleiner oder gleich groß der Menge von Solution Mappings, auf denen die Expression berechnet wird.

Nemo erlaubt zusammengesetzte Expressions und es ist sowohl zeitlich als auch aus Sicht des verwendeten Speichers effizienter, diese zu verwenden, als für jede Teilexpression ein neues Prädikat in Nemo zu definieren. Da jedoch viele SPARQL Expressions, z.B. `NOT EXISTS` sich nicht direkt als Nemo Expression umsätzen lassen, wurde der Ansatz mit einzelnen Prädikaten für jede Teilexpression, aufgrund der Einheitlichkeit, gewählt. In Zukunft ist eine dahingehende Optimierung denkbar (Kapitel 5).

Die einzelnen Expressiontypen der SPARQL-Algebra werden durch Methoden in der `QueryTranslator` Klasse implementiert und in den folgenden Teilkapiteln näher beschreiben.

Zur allgemeinen Veranschaulichung der Implementierungen der Expressiontypen, hier die Deklaration der Funktion, welche eine „oder“ Expression übersetzt:

```
fn translate_or(
    &mut self,
    left: &SolutionExpression,
    right: &SolutionExpression,
    binding: &dyn TypedPredicate
) -> Result<SolutionExpression, TranslateError>
```

[1] dies ist ein anderer `VariableTranslator` als in Unterabschnitt 2.2.5

Dabei werden die Parameter der SPARQL Expression als Nemo Prädikate vom Typ `SolutionExpression`, im Beispiel `left` und `right`, und die Menge an Solution Mappings, für welche die Expression berechnet werden soll, als generisches `TypedPredicate`, im Beispiel `binding`, übergeben. Der generische Parameter ermöglicht es, die Expression Implementierungen auch im Kontext von Multiset-solutions oder Solution-sequences zu verwenden. Wie dies in Nemo Template Language funktioniert, wurde bereits in Unterabschnitt 2.2.2 vorgestellt. Zusätzlich werden häufig noch Parameter spezifisch für die SPARQL Algebra-Expression Operation übergeben. Die Methoden geben jeweils ein Nemo Prädikat vom Typ `SolutionExpression` oder einen Fehler zurück.

Alle spezifischen Übersetzungsmethoden für Expressions werden von der übergeordneten `translate_expression` Methode aufgerufen. Diese führt auch die Rekursion zur Berechnung der Parameter bei verschachtelten Expressions durch. Die korrekte Methode wird in einem großen `match expression` Statement, wobei `expression` die übergebene spargebra `Expression` ist, ausgewählt. Wie auch die spezifischen Expressionmethoden, übernimmt auch die `translate_expression` Methode den `binding` Parameter und gibt ein Nemo Prädikat vom Typ `SolutionExpression` oder einen Fehler zurück.

3.4.1 Variablen

Eine Variablen-Expression evaluiert zu dem Wert, welcher der gegebenen Variable in den gegebenen Bindings zugewiesen ist.

Wie in Abschnitt 3.4 beschrieben, wird eine Variable-Expression in ein Nemo Prädikat übersetzt, welches Tupel, jeweils aus Input-Parametern und dem Ergebnis, *repräsentiert*↗.

Wird z.B. die Expression `?b` auf den Solution Mappings

?a	?b	?c
„a“	1	a
„b“	1	b
„c“	2	c

evaluiert, *repräsentiert*↗ das resultierende Prädikat, die Tupel:

?b	result
1	1
2	2

Als Sonderfall muss betrachtet werden, dass die Variable in einigen Solution Mappings nicht definiert sein kann. Dies kann der Fall sein, wenn die Position, welcher der Variable zugeordnet ist, in einem Tupel der *UNDEF-Marker*↗ ist oder der Variable keine Tupelposition zugeordnet ist. Variablen sind in diesem Zusammenhang, wie in Abschnitt 3.3 beschrieben, SPARQL Variablen, die als Prädikat-Attribute verwendet werden. Die Verwendung einer nicht definierten Variable ist ein Error in SPARQL. Es wird sichergestellt, dass es kein Tupel für die Inputs, welche zu einem Error führen, im *Ergebnis*↗ des Prädikats gibt (siehe Abschnitt 3.2 für den Umgang mit Fehlern).

Da dies die erste vorgestellte konkrete SPARQL zu Nemo Übersetzung ist, wird hier beispielhaft konkret gezeigt, wie eine solche Implementierung vollständig aussieht:

```
fn translate_expression_variable(
    &mut self, var: &Variable, binding: &dyn TypedPredicate
) -> Result<SolutionExpression, TranslateError>
{
    let var_binding = self.sparql_vars.get(var);

    if get_vars(binding).contains(&var_binding)
    {
        let undef = self.undefined_val.clone();
        nemo_def!(
            var(var_binding; @result: var_binding)
            :- binding(??set_contains_var), ~undef(var_binding)
            ; SolutionExpression
        );
        Ok(var)
    }
    else
    {
        let always_error = SolutionExpression::create(
            "always_error", vec![var_binding.clone()]
        );
        Ok(always_error)
    }
}
```

Die Funktion könnte, für das Beispiel auf der vorhergehenden Seite, den folgenden Nemo Code erzeugen:

```
var(?b, ?b) :- p(?a, ?b, ?c), ~undef(?b) .
```

In diesem Code ist der `var` Parameter die SPARQL-Variable `?b` und der `binding` Parameter das Prädikat `p`, dessen Stellen für die Variablen `?a`, `?b` und `?c` stehen. (`?a`, `?b` und `?c` sind nach Definition in Unterabschnitt 1.5.1 die Prädikat-Attribute)

Die Funktion erhält, wie alle Übersetzungsfunktionen, das `QueryTranslator` Objekt `self` und, wie die meisten Expression-Übersetzungen, das Nemo Prädikat mit den Variablen-Bindings, `binding`. Zusätzlich wird die SPARQL Expression Variable `var` vom Typ `Variable`, welcher in der spargebra bibliothek definiert ist, übergeben.

Durch `sparql_vars.get()` wird die SPARQL-Variable in eine entsprechende Nemo Template Language Variable übersetzt (Siehe Abschnitt 3.3).

Das `nemo_def` Makro definiert das resultierende Nemo Prädikat, `var`. Dieses zweistellige Prädikat hat wie alle Expressions den Typ `SolutionExpression`, bei dem die letzte Position das Ergebnis („result“) enthält. Da beide Positionen an `var_binding` *gebunden*↗ werden, haben sie immer den gleichen Wert. Der Wert wird im *Regel-Körper*↗ im `binding` Prädikat *gebunden*↗. Dies macht die Regel „safe“ (siehe Abschnitt 1.5). Das `??set_contains_var` Variablenset umfasst alle Variablen des Binding Prädikats und damit auch die Variable `var_binding` (Siehe Unterabschnitt 2.2.1).

Durch ~undef(var_binding) wird sichergestellt, dass nicht definierte Variablen zu einem Error (fehlendes Tupel) führen. Trotz der Negation führt diese Regel, selbst wenn das Input-Graph-Prädikat vom Ergebnis der SPARQL-Query zyklisch abhängt, nicht zu einem nicht stratifizierten Programm, da das undef Prädikat nicht vom Eingabe-Graph abhängt. (Siehe Abschnitt 1.1 und Unterabschnitt 1.5.2 weshalb dies relevant ist).

Im Falle, dass die SPARQL Variable nicht in den Variablen des binding Prädikats vorkommt, ist das Ergebnis der Übersetzung ein always_error Prädikat, welches in keinem *Regel-Kopf* ↗ vorkommt. Dieses Prädikat *repräsentiert* ↗ die leere Menge und somit einen Fehler für alle Eingaben. Das always_error Prädikat ist einstellig, da es nur eine Position für das Ergebnis hat.

3.4.2 NamedNode und Literal

NamedNode Expression in SPARQL evaluiert zu einer konstanten IRI. Dies lässt sich zu einem einstelligen Nemo Prädikat übersetzen, für welches ein Fakt generiert wird, übersetzen.

Die Implementierung ist recht kurz:

```
fn translate_expression_named_node(
    &mut self, node: &NamedNode
) -> Result<SolutionExpression, TranslateError>
{
    let named_node = SolutionExpression::create(
        "named_node", vec![VarPtr::new("result")]
    );
    nemo_add!(named_node(node.clone()));
    Ok(named_node)
}
```

Außer self wird nur die IRI als NamedNode (Typ aus spargebra) übergeben.

Das resultierende Prädikat named_node vom Typ SolutionExpression wird erstellt und mittels nemo_add Makro aus der Nemo Template Language um einen Fakt erweitert.

Zusätzlich zur abgebildeten Funktion wurde der Trait From<&NamedNode> für Binding implementiert. Binding ist der Typ für Bindings von Prädikatpositionen in der Nemo Template Language und bietet einen Erweiterungspunkt durch Implementierung des From Traits. Dadurch kann ein Objekt vom Typ NamedNode direkt im Nemo Template verwendet werden. (siehe Unterabschnitt 2.2.3)

Literal-Expressions evaluieren zu einem konstanten RDF Literal anstatt einer IRI und können sehr ähnlich wie NamedNode Expressions übersetzt werden.

3.4.3 Effective Boolean Value

Operationen in SPARQL, die einen Wert vom Typ Boolean als Eingabe nehmen, konvertieren nicht boolesche Werte zu einem Boolean mithilfe der „Effective Boolean Value“ Berechnung. Die Übersetzung unterstützt die Effective Boolean Value Berechnung für die Typen bool, str und numeric. Da der Typ zum Zeitpunkt der Übersetzung nicht bekannt ist, werden alle Regeln für die drei Fälle für jede Effective Boolean Value Berechnung generiert. Ein Boolean hat Effective Boolean Value true, wenn er der Wert true ist. Ein String hat den Effective

Boolean Value `true`, wenn er länger als der leere String ist. Eine Zahl hat den Effective Boolean Value `true`, wenn sie nicht Null ist. Ein Beispiel für die Effective Boolean Value Übersetzung bei dem SPARQL-Query Fragment `FILTER(?b)` ist hier dargestellt (xsd Präfix und Kommentare manuell eingeführt):

```
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

% bools
effective_boolean_value(?b, "false"^^xsd:boolean)
   :- var(?b, ?RESULT), ?RESULT = "false"^^xsd:boolean .

effective_boolean_value(?b, "true"^^xsd:boolean)
   :- var(?b, ?RESULT), ?RESULT = "true"^^xsd:boolean .

% str
effective_boolean_value(?b, "false"^^xsd:boolean)
   :- var(?b, ?RESULT), STRLEN(?RESULT) = 0 .

effective_boolean_value(?b, "true"^^xsd:boolean)
   :- var(?b, ?RESULT), STRLEN(?RESULT) > 0 .

% numbers
effective_boolean_value(?b, "false"^^xsd:boolean)
   :- var(?b, ?RESULT), ?RESULT >= 0, ?RESULT <= 0 .

effective_boolean_value(?b, "true"^^xsd:boolean)
   :- var(?b, ?RESULT), ?RESULT > 0 .

effective_boolean_value(?b, "true"^^xsd:boolean)
   :- var(?b, ?RESULT), ?RESULT < 0 .
```

Es gibt jeweils eine eigene Regel für `true` bzw. `false`. Für Zahlen besteht der Fall für `true` aus zwei Regeln, da ein direkter Test auf Gleichheit bzw. Ungleichheit den Typ mit berücksichtigen würde. Es muss sichergestellt sein, dass kein Wert beide Fälle triggert.

Die drei Typen sind die im SPARQL Standard vorgeschriebenen Typen für die Effective Boolean Value Berechnung. Damit kann die Berechnung unter der Annahme, dass Nemo alle vorgeschriebenen Zahlen korrekt unterstützt, als standardkonform bezeichnet werden. Für `NaN` Werte oder falsche Literals wie `äbc"^^xsd:integer` würde der Effective Boolean Value `false`. Jedoch kann Nemo diese Werte ohnehin nicht darstellen. Alle bisher nicht berücksichtigten Werte ergeben einen Fehler, was korrekt abgebildet wird, da in diesem Fall keine der Regeln *zum Tragen*↗ kommt.

Für Subexpressions, von denen bekannt ist, dass sie bereits einen booleschen Wert liefern, muss keine Effective Boolean Value Berechnung durchgeführt werden. Dazu gibt es die Funktion `expression_known_to_be_bool()`, welche mithilfe einer `match` Expression `true` zurückgibt, falls die äußerste Operation den Ergebnistyp Boolean hat oder die Subexpression der konstante Wert `true` oder `false` ist. Die Methode `translate_bool_expression()` verwendet die Funktion `translate_expression()`, `expression_known_to_be_bool()` und `translate_effective_boolean_value()`, um eine Subexpression für eine Operation mit boolescher Eingabe zu übersetzen.

3.4.4 And und Or

Nemo hat eine `OR` und eine `AND` Funktion, welche jedoch nicht direkt verwendet werden können, da sie sich im Fehlerfall nicht korrekt verhalten. SPARQL hat Short-circuit boolesche Operatoren. Beispielsweise ergibt `?a && 0`, wobei `?a` nicht definiert, also ein Error ist, in SPARQL den booleschen Wert `false`. Wenn Fehler als fehlende Tupel dargestellt werden, ist diese Verhalten in Datalog generell schwierig in einer einzigen Funktion umsetzbar. Die Implementierung verwendet stattdessen Standard Datalog Regeln ohne Funktionen.

Hier ist die Übersetzungsfunktion von `OR`:

```
fn translate_or(
    &mut self,
    left: &SolutionExpression,
    right: &SolutionExpression,
    binding: &dyn TypedPredicate
) -> Result<SolutionExpression, TranslateError>
{
    let or = SolutionExpression::create(
        "or",
        nemo_terms![
            left.depend_vars(),
            right.depend_vars(),
            VarPtr::new("result")
        ].vars()
    );
    nemo_add!(
        or(??left, ??right; @result: true)
        :- left(??left; @result: true), binding(??left, ??right, ??other)
    );
    nemo_add!(
        or(??left, ??right; @result: true)
        :- right(??right; @result: true), binding(??left, ??right, ??other)
    );
    nemo_add!(
        or(??both, ??left, ??right; @result: false)
        :-
        left(??both, ??left; @result: false),
        right(??both, ??right; @result: false),
        binding(??both, ??left, ??right, ??other)
    );
    Ok(or)
}
```

Hier ist ein beispielhaftes resultierendes Nemo-Program. `true` und `false` dienen nur zur Veranschaulichung und werden in Wirklichkeit von der Nemo Template Language korrekt in xsd:boolean übersetzt:

```
or(?a, ?b, true) :- left(?a, true), p(?a, ?b, ?c) .
or(?a, ?b, true) :- right(?a, ?b, true), p(?a, ?b, ?c) .
or(?a, ?b, false) :- left(?a, false), right(?a, ?b, false), p(?a, ?b, ?c) .
```

Dabei sind `or`, `left` und `right` Prädikate mit demselben Namen wie im Rust-Code und `p` das `binding` Prädikat. `left` und `right` stehen für die linke und rechte Expression, welche mit OR verknüpft sind.

Die Implementierung folgt folgender Aussage: „a oder b“ ist genau dann wahr, wenn „a“ wahr ist oder „b“ wahr ist und genau dann falsch, wenn „a“ und „b“ beide falsch sind. Ist z.B. „a“ falsch und „b“ ein Fehler, ist „a oder b“ weder wahr noch falsch - also ein Fehler.

Die Übersetzungsfunktion erstellt zunächst ein neues Prädikat `or`, welchem dann drei Regeln hinzugefügt werden.

Das `or`-Prädikat hat Positionen für Abhängigkeiten von linker und rechter Subexpression und, wie alle `SolutionExpression` Prädikate, das Ergebnis als letztes Prädikat-Attribut. Diese Positionen werden mit dem `nemo_terms!` Makro gemerged. Dieses entfernt Duplikate, ohne die Reihenfolge zu ändern. Die `depend_vars()` Methode gibt für `SolutionExpression` alle Variablen bis auf die letzte, welche für das Ergebnis ist, zurück.

Die drei Regeln werden mit dem `nemo_add!` Makro der Nemo Template Language zum `or` Prädikat hinzugefügt. Die ersten beiden Regeln stehen für: „a oder b“ ist genau dann wahr, wenn „a“ wahr ist oder „b“ wahr ist. Die dritte Regel steht für „a oder b“ ist genau dann falsch, wenn „a“ und „b“ beide falsch sind.

Das Binding Prädikat in jeder Regel schränkt die möglichen Inputkombinationen ein. Dies ist wichtig, falls die linke und rechte Teilexpression nicht von den gleichen Variablen abhängen. Dies würde in den ersten beiden Regeln zum Verstoß gegen die Safety Bedingung (Abschnitt 1.5) und in der letzten Regel, unter Umständen, zu einem großen *Ergebnis*↗ führen, da zwei Variablen, die nur auf einer Seite jeweils vorkommen, im Ergebnis alle möglichen Kombinationen liefern würden. Es ist darauf geachtet worden, alle möglichen Kombinationen, mit denen Variablen in Prädikaten auftauchen können, mit Variablen Sets (beginnend mit `??`, siehe Unterabschnitt 2.2.1) zu berücksichtigen.

Die zugrundeliegende Aussage ist durch die De-morganschen Gesetze inspiriert und kann auch mit diesen in eine Aussage für `AND` überführt werden: „a und b“ ist genau dann falsch, wenn „a“ falsch ist oder „b“ falsch ist und genau dann wahr, wenn „a“ und „b“ beide wahr sind. Diese Aussage kann für einer Übersetzung von `AND` ganz ähnlich zu `OR` verwendet werden. Eine formale Herleitung wäre nur möglich, nachdem einige Gesetze der in SPARQL verwendeten Logik mit „Wahr“, „Falsch“ und „Error“ eingeführt würden. Die Korrektheit der Aussagen und Übersetzungsimplementierung kann durch Testen aller neun im SPARQL Standard festgelegten Möglichkeiten[2] gezeigt werden.

Es ist die korrekte Implementierung der Effective Boolean Value Berechnung (Unterabschnitt 3.4.3) der linken und rechten Teilexpression zu beachten.

3.4.5 Not

Nemo hat eine NOT Funktion, die boolesche Negation berechnet. Diese wird zur Übersetzung der Not Expression aus SPARQL verwendet. Allerdings ist zu beachten, die Effective Boolean Value Berechnung (Unterabschnitt 3.4.3) korrekt durchzuführen.

Beispiel Nemo-Code:

```
boolean_not(?a, ?b, NOT(?result)) :- inner(?a, ?b, ?result) .
```

[2] https://www.w3.org/TR/sparql11-query/#truthTable

Jede, auch zusammengesetzte, (Sub-) Expression wird, der Einfachheit halber, zu einem einzigen Prädikat übersetzt. Deswegen kann Not nicht durch ein simples zweistelliges Prädikat, welches die Warheitstabelle für Not ({(`true`, `false`), (`false`, `true`)}) *repräsentiert*↗, dargestellt werden und die Abhängigkeiten der inneren Expression müssen „durchgereicht" werden.

Falls die innere Expression für eine Kombination von Eingaben einen Error ergibt, gibt es kein Tupel im *Ergebnis*↗ des Prädikats und somit auch kein Tupel im *Ergebnis*↗ des `boolean_not` Prädikats. Der Fehler wird somit korrekt propagiert.

3.4.6 In

Eine `In` Expression in SPARQL testet, ob das Ergebnis einer Expression gleich einem Ergebnis einer Expression aus einer gegebenen Liste von Expressions ist. z.B. `FILTER(1+1 in (1, ?a, 3))` lässt nur Ergebnisse mit `?a` gleich `2` zu.

Eine `In` Expression ist genau dann Wahr (Wahr-Fall) wenn einer der Expressions aus der Liste das gleiche Ergebnis (kein Error) wie die separat gegebene Expression hat. Eine `In` Expression ist genau dann falsch (Falsch-Fall) wenn alle Expressions aus der Liste ein anderes Ergebnis wie die separat gegebene Expression und keinen Error liefern. Fehler können also ignoriert werden, wenn der Wahr-Fall dennoch zutrifft. Da nur mit Ergebnissen von Expressions gerechnet wird, welche immer definiert sind, muss der Fall undefinierter Variablen nicht berücksichtigt werden. Dies ist ein sehr ähnliches verhallten wie bei AND und OR (Unterabschnitt 3.4.4) und die Übersetzung findet auch ähnlich statt. Einziger unterschied, ist das AND und OR in der SPARQL-Algebra nur eine linke und rechte Teilexpression haben während `In` eine beliebige Anzahl an Teil-Expressions erlaubt. Dies stellt jedoch kein Problem dar, da die Nemo Template Language es erlaubt, Regeln innerhalb einer Rust-Schleife zu erzeugen (für Wahr-Fall) und auch mittels `nemo_atoms!` Makro (Unterabschnitt 2.2.4) Teile einer Regel in einer Rust-Schleife zu erzeugen. Konkret ist eine `In` Expression laut Standard äquivalent zur `oder` Verknüpfung von Tests, wobei jeder Test das Ergebnis der separat gegebenen Expression mit dem Ergebnis einer Expression aus der Liste vergleicht.

Hier ist eine komplette Übersetzung einer ASK-Query mit `In` Expression (xsd Datentypen zur Lesbarkeit entfernt, int und boolean sind in Realität typisiert):

SPARQL:

```
ASK {
    FILTER(2 in (1, ?a, 3))
}
```

Nemo:

```
dummy(arity_zero) .
bgp(arity_zero) :- dummy(arity_zero) .


literal(2) .
literal_1(1) .
literal_2(3) .

in_expression(false)
:- bgp(arity_zero),
   literal(?RESULT),
```

```
    literal_1(?RESULT_1),
    always_error(?RESULT_2),
    literal_2(?RESULT_3),
    ?RESULT_1 != ?RESULT, ?RESULT_2 != ?RESULT, ?RESULT_3 != ?RESULT .

in_expression(true)
:- bgp(arity_zero), literal(?RESULT), literal_1(?RESULT) .

in_expression(true)
:- bgp(arity_zero), literal(?RESULT), always_error(?RESULT) .

in_expression(true)
:- bgp(arity_zero), literal(?RESULT), literal_2(?RESULT) .


filter(arity_zero) :- bgp(arity_zero), in_expression(true) .

dummy_1(0) .
ask(true) :- filter(arity_zero) .
ask(false) :- dummy_1(0), ~filter(arity_zero) .
@output ask .
```

Zum Verständnis des Nemo Beispiel-Programms hier zunächst einige relevante Informationen aus anderen Kapiteln: Im code wird die konstante `arity_zero` verwendet. Dieses wird von der Nemo Template Language generiert, wenn ein Parameter Template für einen konkreten Fall zu keinen Parametern führt. Führt ein Template zu keinem *Regel-Körper*↗, wird ein `dummy` Prädikat verwendet. Siehe Unterabschnitt 2.2.5 für weitere Informationen. `bgp` steht für Basic-Graph-Pattern und wird in Unterabschnitt 3.7.2 näher erläutert. Die Beispielquery enthält einen impliziten leeren Basic-Graph-Pattern.

Die erste und dritte Regel von `in_expression` kommen nicht *zum Tragen*↗, da das `always_error` Prädikat in keinem *Regel-Kopf*↗ vorkommt. So auch die zweite und vierte Regel, weil 2 nicht gleich 1 bzw. 3 ist. Damit *repräsentiert*↗ `in_expression` und somit auch `filter` die leere Menge und die letzte Regel, mit `ask(false)`, kommt *zum Tragen*↗ was zum korrekten Ergebnis der Query führt: `false`.

Vor dem Hintergrund dieses Beispiels sind einige von SPARQL geforderten Eigenschaften zu erkennen:

- Eine `In` Expression mit leerer liste ist gültig: Im Beispiel würde der Falsch-Fall bei leerer liste zu `in_expression(false) :- bgp(arity_zero), literal(?RESULT)`. Die in Expression würde also zu `false` evaluieren. Dies ist korrekt. Im Standard ist explizit das Beispiel `2 IN ()` ist `false` gegeben. Jedoch sollte die `In` Expression auch zu `false` evaluieren, wenn die Liste leer und die Expression selbst ein Error ist, was mit der derzeitigen Übersetzung nicht korrekt funktioniert.
- Zur Implementierung der Vergleiche wird eine strengere Form von Gleichheit in Nemo verwendet als im SPARQL Standard vorgesehen. Ein Beispiel aus dem Standard besagt `2 IN (<http://example/iri>, ßtr", 2.0)` ist `true`. `2` matcht jedoch nicht `2.0` mit der verwendeten vorm von Gleichheit in Nemo. Näheres dazu in Unterabschnitt 3.4.10.

Außer der `In` Expression gibt es auch die `Not In` Expression. Diese ist äquivalent zu einer `In` Expression mit Negation. Eine `Not In` Expression wird bereits in spargebra zu einer `In` und `Not` Expression konvertiert und muss nicht separat berücksichtigt werden.

3.4.7 If

Eine `If`-Expression hat drei Sub-Expressions. Eine Condition-Expression, die auswählt, welche der anderen zwei Sub-Expressions zur Berechnung des Ergebnisses verwendet wird, eine Wahr-Expression und eine Falsch-Expression, welche jeweils basierend auf dem Ergebnis der Condition-Expression verwendet werden.

Die `If`-Expression kann durch zwei Regeln in Nemo umgesetzt werden. Eine Regel, die Ergebnisse für den Wahr-Fall erzeugt und eine für den Falsch-Fall.

So sieht das Fragment der `If`-Expression für die Expression `IF(?c, ?t, ?f)` z.B. aus (xsd Präfix nachträglich eingeführt):

```
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

if_expression(?c, ?t, ?f, ?RESULT)
:- bgp(?a, ?c, ?t, ?f),
   effective_boolean_value(?c, "true"^^xsd:boolean),
   var_1(?t, ?RESULT) .

if_expression(?c, ?t, ?f, ?RESULT)
:- bgp(?a, ?c, ?t, ?f),
   effective_boolean_value(?c, "false"^^xsd:boolean),
   var_2(?f, ?RESULT) .
```

Wie bei anderen Expressions *repräsentiert*↗ das `bgp` Prädikat die Variablen-Bindings und stellt sicher, dass die Regeln save (Abschnitt 1.5) sind. Die `If`-Expression hängt nicht von der Variable `?a` ab. Diese ist im konkreten Beispiel dennoch Teil des Basic-Graph-Patterns. `var_1` und `var_2` sind die Prädikate der Sub-Expressions, im konkreten Beispiel ist der erste und zweite Parameter für sie immer identisch (siehe Unterabschnitt 3.4.1).

SPARQL gibt vor, dass die `If`-Expression einen Error ergibt, wenn die Condition-Expression einen Fehler verursacht. Eine Parameterkombination, welche einen Fehler bewirkt, ist nicht im *Ergebnis*↗ von `effective_boolean_value` (siehe Unterabschnitt 3.4.3) und somit kommt keine der beiden Regeln *zum Tragen*↗ und das Ergebnis ist korrekterweise ein Error. Weiterhin gibt SPARQL vor, dass die Sub-Expression, die nicht für das Ergebnis benötigt wird, nicht berechnet wird. Die Übersetzung berechnet jedoch beide Expressions. Dies führt zu einem identischen Ergebnis da die Berechnung keinen Sideffects hat, vorausgesetzt ein Fehler in der nicht zu evaluierenden Sub-Expression verursacht keinen Error im Ergebnis. Dies ist gegeben, da die entsprechende Regel ohnehin nicht *zum Tragen*↗ kommt.

3.4.8 Bound

Bound ist eine SPARQL Expression, die testet, ob eine Variable in einem Solution Mapping definiert ist, z.B. `bound(?a)` ist wahr, wenn ?a einen Wert hat. Nach dem in Abschnitt 3.1 beschriebenen Umgang mit Unbound-Werten, muss zur Implementierung der Bound expression geprüft werden, ob der entsprechende Wert der *UNDEF-Marker*↗ ist. In Nemo Template Language kann dies wie folgt aussehen:

```
nemo_def!(
    bound(nemo_var; @result: true) :- binding(??vars), ~unbound(nemo_var)
    ; SolutionExpression
```

```
);
nemo_add!(
    bound(nemo_var; @result: false) :- binding(??vars), unbound(nemo_var)
);
```

Das `nemo_def!` Makro erstellt ein neues Prädikat `bound` mit Regel für den Wahr-Fall und mit `nemo_add!` wird eine weitere Regel für den Falsch-Fall hinzugefügt (Makros werden in Unterabschnitt 2.2.4 beschrieben). `binding` stellt den Wert der Variable zur Verfügung. `nemo_var` ist die Nemo Template Language Variable zur SPARQL-Variable von welcher geprüft werden soll, ob sie an einen Wert gebunden ist. Das *Ergebnis*↗ des `unbound` Prädikats enthält nur den *UNDEF-Marker*↗. Das `unbound` Prädikat *repräsentiert*↗ den *UNDEF-Marker*↗[3]. Der Wahr-Fall ergibt `true` für alle Werte, die nicht der *UNDEF-Marker*↗ sind. Der Falsch-Fall ergibt falsch für den *UNDEF-Marker*↗ falls dieser für die Variable vorkommt.

Die zweite Regel wird, mittels einer if-Bedinung in Rust, nur generiert, falls die Variable nicht bereits als „immer gebunden" markiert ist. (siehe Abschnitt 2.2 für automatisches Tracking von Eigenschaften) Ein weiterer Sonderfall stellt dar, wenn die Variable für die Bound getestet werden soll, nicht in den Stellen des `binding` Prädikats auftaucht. In diesem Fall wäre die erste Regel nicht safe (Abschnitt 1.5). Es wird, ähnlich wie bei Expression-Variablen (Unterabschnitt 3.4.1), in diesem Fall die Expression zu einem extra Prädikat `never_bound` welches `false` *repräsentiert*↗ übersetzt[4].

3.4.9 Exists

Neben MINUS (Unterabschnitt 3.7.7) ist eine wesentliche vorm von Negation in SPARQL `FILTER NOT EXISTS`. Beispielsweise findet folgende Query alle Dinge, die ein `ex:a` aber kein `ex:b` für die Eigenschaft `ex:p` haben:

```
prefix ex: <https://example.org/>

SELECT DISTINCT ?a {
    ?a ex:p ex:a
    FILTER NOT EXISTS {
        ?a ex:p ex:b
    }
}
```

Ähnlich wie bei `In` (Unterabschnitt 3.4.6) und `Not In` ist laut SPARQL Standard `NOT EXISTS` äquivalent zu `EXISTS` mit angeschlossener Negation. Die spargebra Bibliothek führt diese Substitution automatisch aus und kennt keine separate Algebra-Operation für `NOT EXISTS`, sondern nur eine für `EXISTS`. Es ist jedoch bekannt, dass allgemeine Negation in Datalog zu Problemen führen kann, weshalb Nemo stattdessen stratifizierte Negation unterstützt (Unterabschnitt 1.5.2). Die Schwierigkeit liegt jedoch nicht in der Not-Operation, welche durch einen einfachen Aufruf der `NOT` Funktion in Nemo übersetzt wird (Unterabschnitt 3.4.5), sondern im Fall, dass die `EXISTS` Expression das Ergebnis `false` liefert. Dieser Fall spielt jedoch ein vielen Fällen, in denen `EXISTS` ohne `NOT` verwendet wird, keine Rolle. Bei einem Nemo-Programm, in dem eine SPARQL Query vorkommt, deren Eingabegraph zyklisch vom Ergebnis der Query abhängt (siehe Abschnitt 1.1) ist bei `FILTER NOT EXISTS` in der Query ein nicht stratifiziertes Programm, jedoch bei nur `FILTER EXISTS` nicht zwangsläufig

[3] Formal korrekt *repräsentiert*↗ das Prädikat eine einelementige Menge mit einem ein-Tupel: `{(x)}` mit dem *UNDEF-Marker*↗ x

[4] Formal korrekt *repräsentiert*↗ das Prädikat eine einelementige Menge mit einem ein-Tupel: `{(false)}`

ein nicht stratifiziertes Programm zu erwarten. Deswegen ist die Übersetzung von `EXISTS` unterteilt in eine Funktion für den positiven Spezialfall `translate_positive_exists()` und eine allgemeine Funktion `translate_exists()` welche die `translate_positive_exists()` Funktion und stratifizierte Negation verwendet, um das Ergebnis für den Falsch-Fall zu vervollständigen. Die `translate_exists()` Funktion ist sehr einfach und kann hier komplett abgebildet werden:

```
fn translate_exists(
    &mut self, pattern_solution: &SolutionSet, binding: &dyn TypedPredicate
) -> Result<SolutionExpression, TranslateError>
{
    let partial_exists = self.translate_positive_exists(
        pattern_solution, binding
    )?;
    nemo_def!(
        exists(??vars; @result: true)
        :- partial_exists(??vars; @result: true)
        ; SolutionExpression
    );
    nemo_add!(
        exists(??vars; @result: false)
        :- binding(??vars, ??other), ~partial_exists(??vars; @result: true)
    );
    Ok(exists)
}
```

Die Funktion nimmt neben dem üblichen `self` und `binding` Parameter die `pattern_solution` als Eingabe. Die `pattern_solution` ist ein Nemo Prädikat welches das *Query-Ergebnis*↗ des Exists-Pattern *repräsentiert*↗. Reihenfolge und Multiplizität ist jedoch für den Exists-Pattern nicht relevant. Deswegen ist die `pattern_solution` vom einfachen Typ `SolutionSet` welcher keine besonderen Prädikatpositionen definiert (Abschnitt 2.3).

Die Funktion verwendet das `partial_exists` Prädikat welches durch die Methode `translate_positive_exists()` erstellt wird und bereits das Korrekte *Ergebnis*↗ *repräsentiert*↗, wenn man sich auf Tupel mit `true` in als Ergebnis beschränkt. Alle Ergebnisse in `partial_exists` sind `true`. Die korrekten Wahr-Fall-Ergebnisse werden durch die erste Regel, im `nemo_def` Makro, einfach kopiert. Für den Falsch-Fall werden, unter Verwendung von stratifizierter Negation (Unterabschnitt 1.5.2), in der zweiten Regel, im `nemo_add!` Makro, alle Eingaben mit dem Ergebnis `false` hinzugefügt, die nicht bereits durch die erste Regel mit `true` hinzugefügt wurden. Das VarSet `??other` wird benötigt, um alle Variablen, die im `binding` vorkommen, von denen jedoch die `EXISTS` Expression nicht abhängt abzudecken. (Siehe Unterabschnitt 2.2.1)

Die eigentliche Übersetzung in `translate_positive_exists()` gehört zu einer Kategorie von Funktionen, die ähnlich zu einem Join funktionieren und wird in Unterabschnitt 3.7.3 mit vorgestellt und verglichen. Die Funktion ist durch das Behandeln von nicht gebundenen Variablen recht komplex. Die eigentliche Funktionsweise kann jedoch bereits hier an einem Beispiel ohne nicht gebundene Variablen nachvollzogen werden. Das Beispiel ist das Nemo-Programm von der Übersetzung der oben vorgestellten `FILTER NOT EXISTS` Query. (zur besseren Lesbarkeit wurde die Formatierung manuell angepasst und die Präfixe `ex:` und `xsd:` eingeführt und für `true` und `false` der `xsd:boolean` Type entfernt)

```
@prefix ex: <https://example.org/> .

bgp(?a) :- input_graph(?a, ex:p, ex:a) .
bgp_1(?a) :- input_graph(?a, ex:p, ex:b) .

partial_exists(?a, true) :- bgp(?a), bgp_1(?a) .

exists(?a, true) :- partial_exists(?a, true) .
exists(?a, false) :- bgp(?a), ~partial_exists(?a, true) .
boolean_not(?a, NOT(?b)) :- exists(?a, ?b) .

filter(?a) :- bgp(?a), boolean_not(?a, true) .
projection(?a) :- filter(?a) .
@output projection .
```

Man kann beobachten, dass `EXISTS` in diesem Beispiel im Wesentlichen durch den einfachen Join `bgp(?a), bgp_1(?a)` implementiert wurde. Hier noch einige Dinge, die beim Betrachten des Nemo-Programmbeispiels nicht offensichtlich sind: `bgp` steht für Basic-Graph-Pattern, was später in Unterabschnitt 3.7.2 beschrieben wird. Der Ursprung des Variablen-Namen `?b` ist auf den ersten Blick überraschend, dieser stammt explizit aus dem Nemo Template Language Template, da die Nemo Template Language keine implizit benannten Variablen in Funktionsaufrufen unterstützt (Unterabschnitt 2.2.4).

Die Unterscheidung wann die `translate_positive_exists()` Funktion direkt genutzt wird, passiert in der `translate_pattern()` Funktion (Abschnitt 3.7). Dort gibt es einen Sonderfall für `FILTER EXISTS` in allen anderen Fällen wird die generische `translate_exists()` Funktion verwendet.

Im SPARQL Standard wird das Verwenden von Variablen aus dem umgebenden Scope in der `EXISTS` Expression vollständig unterstützt. Dies ist eine wesentliche Abweichung vom SPARQL Standard bei der Übersetzung. Durch den grundlegenden Ansatz, dass die Operationen bottom-up nach der SPARQL-Algebra Darstellung bearbeitet werden, werden Subexpressions in der `EXISTS` Expression und die der umgebende Scope unabhängig voneinander übersetzt. So führt beispielsweise ein Test `Bound(?x)` innerhalb eines `EXISTS` Filters zum Ergebnis `false`, wenn `?x` nur im umgebenden Scope gebunden wird. Dieses Problem wird in Kapitel 5 noch einmal aufgegriffen.

3.4.10 Vergleichsoperatoren

SPARQL erlaubt es Werte mit Vergleichsoperatoren wie Größer-Als, „>“, oder Ungleich, „!=“, zu vergleichen. Nemo behandelt unterschiedliche Typen auf unterschiedliche Weise, was einer kurzen Übersetzung entgegenwirkt. Die Übersetzung der Vergleichsoperatoren umfasst über 150 Zeilen. Die Implementierung lässt sich in drei Teile untergliedern:

- Eine Hilfsfunktion `translate_comparison()`, die, mittels Nemo Template Language, Nemo code generiert, in den, als Parameter gegebene, Nemo Template Language Fragmente eingearbeitet werden.
- Eine spezielle Funktion `translate_equal()`, die einen = Vergleich aus SPARQL übersetzt.
- Ein Match-Statement in Rust, welches für die jeweiligen Elemente der Expression-Enumeration aus spargebra mittels der Hilfsfunktionen die korrekte Operation durchführt.

Die grundlegende Herangehensweise ist es, für jeden Typen eine Regel für den Falsch-Fall und eine für den Wahr-Fall zu generieren. Die finale Übersetzung muss dann immer alle Regeln beinhalten, da mögliche Einschränkungen des Typs zur Übersetzungszeit nicht bekannt sind.

Die `translate_comparison()` Funktion unterstützt die Typen Numeric, Boolean und, String und keine Vergleiche für zwei unterschiedliche dieser drei Typen. Damit ist die Funktion fast Standard konform, da der Standard nur zusätzlich mindestens Unterstützung für Date-Time fordert. Da jedoch Nemo diesen Typ nicht unterstützt, wurde auf dessen Implementierung hier verzichtet. Eine Implementierung sehr ähnlich zu der in Unterunterabschnitt 3.4.13 vorgestellten wäre jedoch denkbar.

Numerische Vergleiche werden direkt zu Vergleichsoperatoren in Nemo übersetzt. Boolean Werte werden zunächst mit der in Nemo verfügbaren `INT` Funktion zu integer konvertiert und dann mit denselben Vergleichsoperatoren wie numerische Werte verglichen. Die Funktion nimmt dafür die Parameter `ture_op` und `false_op`. Strings werden mit der `COMPARE` Funktion in Nemo verglichen und das Ergebnis mit einem gegebenen Wert `str_compare` verglichen. Bei Gleichheit ist das Ergebnis des Vergleiches insgesamt `str_compare_for` (also `true` bzw. `false`), bei Ungleichheit das Gegenteil von `str_compare_for`. Folgende Tabelle zeigt alle 4 Parameter für die jeweiligen SPARQL Operatoren:

SPARQL op	true_op	false_op	str_compare	str_compare_for
Greater	>	<=	1	true
GreaterOrEqual	>=	<	-1	false
Less	<	>=	-1	true
LessOrEqual	<=	>	1	false

Hier ein Beispiel, zu welchen Operationen ein Größer-Als-Vergleich (Greater) von ?x und ?y aus SPARQL übersetzt wird:

Nemo Expression	Expression für
?x > ?y	true
?x <= ?y	false
INT(?x) > INT(?y) and datatypes are xsd:boolean	true
INT(?x) <= INT(?y) and datatypes are xsd:boolean	false
COMPARE(?x, ?y) = 1	true, da `str_compare_for` true ist
COMPARE(?x, ?y) != 1	false (= not `str_compare_for`)

Es ist wichtig, dass die Regeln für Boolean-Werte den Typ prüfen, da sonst Fälle, in denen der Vergleich sowohl `true` als auch `false` ist. z.B. `COMPARE("3", "12") = 1` führt zu true, aber `INT("3") <= INT("12")` führt zu false. Die anderen Regeln ergeben ohnehin einen Fehler bei falschem Typ.

Die Gleichheitsoperation = in SPARQL, wird, für alle nicht numerischen Typen, zur Gleichheitsoperation =, bzw. != im Falsch-Fall, in Nemo übersetzt. Diese Operation beachtet jedoch in Nemo keine Typkonvertierungen, wie dies bei den anderen Vergleichsoperatoren der Fall ist. Deswegen wird Gleichheit von ?x und ?y für numerische Typen wie Folgt übersetzt:

- **Wahr-Fall:** Eine Regel, die sowohl `?x >= ?y` als auch `?x <= ?y` prüft
- **Falsch-Fall:** Zwei Regeln, entweder `?x < ?y` oder `?x > ?y`

`SameTerm` in SPARQL ist eine weitere Gleichheitsoperation. Diese wird mit `fullStr(?x) = fullStr(?y)` für den Wahr-Fall bzw.`fullStr(?x) != fullStr(?y)` für den Falsch-Fall übersetzt.

Der Ungleich-Operator wird bereits von spargebra zu einem Gleichheitsoperator und Negation transformiert und hat keine eigene Implementierung.

Es ist anzumerken, dass die hier vorgestellte Übersetzung dennoch keine SPARQL konforme Implementierung ergibt. z.B. werden in Nemo Werte normalisiert, was dazu führt, dass unterschiedliche Werte als gleich befunden werden. Auch beachtet die Typkonvertierung von numerischen Werten in Nemo nicht die im SPARQL Standard vorgeschriebenen „numeric type promotion“ und „subtype substitution“ Regeln, sondern vergleicht unterschiedliche Typen, indem beide werde zunächst zu Double konvertiert werden. Das schwerwiegendste Problem stellt jedoch dar, dass Nemo, in der verwendeten Version, keine Werte vom Typ xsd:decimal vergleichen kann und dies der Standardtyp ist, wenn in SPARQL eine Zahl mit Komma, wie z.B. `0.9`, ohne expliziten Typ verwendet wird. Zwar wäre eine automatische Konvertierung von `decimal`-Werten in der Literal-Implementierung (Unterabschnitt 3.4.2) möglich, jedoch ist es sicher nicht komplex den `decimal`-Typ in Nemo zu unterstützen.

3.4.11 Coalesce

Die `COALESCE`-Expression in SPARQL ergibt das erste Ergebnis aus einer Liste von Expressions, welches kein Error ist. `COALESCE(1/0, 2, 3)` ergibt beispielsweise `2`. Dieses Beispiel sei hier ergänzt um Prädikate für bereits übersetzte Subexpressions. Jede der drei Subexpressions wird im Beispiel in ein einstelliges Prädikat übersetzt. `1/0` wird zum Prädikat `op` übersetzt (siehe Unterabschnitt 3.4.12). Da die Division einen Fehler ergibt *repräsentiert*$^{\nearrow}$ `op` die leere Menge (Siehe Abschnitt 3.2 für den Umgang mit Fehlern). `literal_2` und `literal_3` *repräsentieren*$^{\nearrow}$ jeweils 2 bzw. 3[5]. Folgendes wäre eine initiale Implementierung, welche die drei Ergebnisse zusammen führt:

```
coalesce(?RESULT) :- op(?RESULT) .
coalesce(?RESULT) :- literal_2(?RESULT) .
coalesce(?RESULT) :- literal_3(?RESULT) .
```

Da die Reihenfolge der Parameter nicht egal ist, lässt sich `COALESCE` nicht durch gleichförmige Regeln für jeden Parameter, wie in dieser initialen Implementierung, übersetzen. Die tatsächliche Implementierung fügt jeder Regel noch eine Bedingung hinzu, dass keine der vorherigen Expressions bereits ein Nicht-Error-Ergebnis geliefert hat:

```
coalesce(?RESULT)
:- bgp(arity_zero), op(?RESULT) .

coalesce(?RESULT)
:- bgp(arity_zero), literal_2(?RESULT),
   ~op(?RESULT_1) .

coalesce(?RESULT)
:- bgp(arity_zero), literal_3(?RESULT),
   ~op(?RESULT_1), ~literal_2(?RESULT_2) .
```

[5] Formal die einelementige Menge aus dem ein-Tupel der Zahl, z.B. {(2)}.

Damit hat nur die Regel, welche als Erstes keinen Error verursacht (im Beispiel die zweite) einen Effekt auf das Ergebnis. Alle vorherigen Regeln (im Beispiel die erste) evaluieren definitionsgemäß zu einem Error und beeinflussen somit nicht das Ergebnis. Alle späteren Regeln enthalten das Prädikat der relevanten Regel in negierter vorm und haben somit keinen Einfluss auf das Ergebnis. Wenn alle Expressions einen Error verursachen, kommt keine Regel *zum Tragen*↗ und das Ergebnis ist ein Error.

Wie bei vorherigen Expression-Übersetzungen werden die Variablen-Bindings, das Prädikat `bgp` im Beispiel, benötigt damit alle Regeln safe (Abschnitt 1.5) sind, auch wenn Subsexpressions von unterschiedlichen Variablen abhängen.

Die Implementierung generiert die Regeln mittels eines for-Loops in Rust. Die negierten Atome werden mittels des `nemo_atoms!` Makro (Unterabschnitt 2.2.4) in jeder Iteration akkumuliert.

Es ist anzumerken, dass eine so übersetzte `COALESCE`-Expression ein nicht stratifiziertes Nemo-Programm bewirkt, wenn der Eingabegraph vom Ergebnis der Query abhängt. Es gellten die allgemeinen Überlegungen zum Darstellen von Fehlern durch fehlende Tupel (Abschnitt 3.2).

3.4.12 Arithmetische Operatoren

Arithmetische Operatoren in SPARQL (Add, Subtract, Multiply und Divide) werden direkt zu den jeweiligen Operatoren in Nemo übersetzt. Unäre Operatoren (UnaryMinus und UnaryPlus) werden übersetzt wie Subtraktion von bzw. Addition zu 0.

Die Implementierung ist sehr einfach und kann deswegen, hier als weiteres Beispiel wie eine SPARQL-Feature-Implementierung aussieht abgebildet werden:

```
fn translate_binary_operator(
    &mut self,
    l: Binding,
    r: Binding,
    result: Binding,
    left_solution: &SolutionExpression,
    right_solution: &SolutionExpression,
    binding: &dyn TypedPredicate
) -> Result<SolutionExpression, TranslateError>
{
    nemo_def!(
        op(??vars, ??left, ??right; @result: result) :-
            binding(??vars, ??left, ??right, ??other),
            left_solution(??vars, ??left; @result: l),
            right_solution(??vars, ??right; @result: r)
            ; SolutionExpression
    );
    Ok(op)
}
```

Die Funktion `translate_binary_operator()` übersetzt einen beliebigen binären Operator. Der eigentliche Operator wird, wie unten gezeigt, als `result` Parameter, in Form eines Nemo Template Language Fragmentes, der Funktion übergeben. Die beiden Werte für die Berechnung kommen vom Ergebnis der Linken bzw. Rechten Subexpression (`left_solution`

bzw. `right_solution` Prädikat). Die tatsächlichen Variablen dafür werden als Parameter übergeben, damit diese sowohl bei der Operation als auch im `result` Parameter verwendet werden. (Siehe Unterabschnitt 2.2.1 für mehr Information zu den mit `??` beginnenden Variablen Sets im Code-Beispiel). Wie die Funktion aufgerufen wird, wird hier am Beispiel von Addition gezeigt:

```
Expression::Add(left, right) => {
    let left_solution = self.translate_expression(left, binding)?;
    let right_solution = self.translate_expression(right, binding)?;
    let l = nemo_var!(l);
    let r = nemo_var!(r);
    self.translate_binary_operator(
        l.clone(), r.clone(), l + r,
        &left_solution, &right_solution, binding
    )
}
```

Das Code-Fragment ist ein Ausschnitt aus der Rust Match-Expression, welches sich in der `translate_expression()` Funktion (Siehe Abschnitt 3.4) befindet. Zunächst werden die Subexpressions übersetzt. Dann neue Variablen für linkes und rechtes Ergebnis erzeugt. Schließlich wird die `translate_binary_operator()` Funktion aufgerufen. Die eigentliche Operation wird durch `l + r` definiert. Die Nemo Template Language beinhaltet Überladungen für arithmetische Operatoren. (Siehe Unterabschnitt 2.2.3)

Ein Fehler in einer Subexpression bewirkt das Fehlen der Parameterkombination im jeweiligen *Ergebnis*↗ des Prädikats, zu welchem die Subexpression übersetzt wird. Dadurch gibt es auch diese Parameterkombination nicht im Ergebnis der Operation, was gleichbedeutend mit einem Error ist. Das bedeutet Errors propagieren korrekt.

Es ist anzumerken, dass es Unterschiede beim Umgang mit unterschiedlichen Typen bei den Operatoren in Nemo gegenüber SPARQL gibt (siehe auch Unterabschnitt 3.4.10 dazu). Damit sind die Operatoren nicht standardkonform. Besonders gravierend sind:

- Limitierte Unterstützung für xsd:decimal (dieser ist der Standardtyp in SPARQL; die FLOAT Funktion in Nemo unterstützt diesen Typ bereits und es gibt in der verwendeten Version automatische Konvertierung für Integer vom Typ xsd:decimal)
- Nicht Standardkonformer Rückgabetyp. Insbesondere 9 / 10 ergibt 0, obwohl explizit im SPARQL Standard über den Rückgabetyp der Division gesagt wird „xsd:decimal if both operands are xsd:integer“

3.4.13 Funktionen

Das Ziel von Nemo ist es, die SPARQL Funktionen direkt zu unterstützen. In der verwendeten Nemo Version ist dies jedoch noch nicht vollständig der Fall. Einige nicht in Nemo unterstützte Funktionen können mithilfe anderer Funktionen implementiert werden, für einige wurde dies im Rahmen dieser Arbeit umgesetzt. Insbesondere hat Nemo in der verwendeten Version keine Unterstützung für Datum und Uhrzeit, es konnten jedoch einige derartige Funktionen durch String Manipulationsfunktionen übersetzt werden.

Direkt unterstützte Funktionen

Die meisten Funktionen in SPARQL werden direkt mit einer Funktion in Nemo übersetzt. Die eigentliche Übersetzung erfolgt dabei mithilfe der `function()` Methode des `QueryTranslator` Objektes:

```
fn function(
    &mut self,
    expressions: &Vec<SolutionExpression>,
    binding: &dyn TypedPredicate,
    func: &str
) -> Result<SolutionExpression, TranslateError>
{
    let call = Call::new(
        func,
        nemo_terms!(expressions => SolutionExpression::result_var).bindings()
    );
    nemo_def!(
        solution(
            nemo_terms!(expressions => SolutionExpression::depend_vars, call)
        ) :- {expressions}, {binding}
        ; SolutionExpression
        ; &func.to_lowercase()
    );
    Ok(solution)
}
```

Die Funktion nimmt neben den bisher bekannten Parametern `self` und `binding`. Die Nemo Funktion als `&str` und die Parameter, mit denen diese aufgerufen werden soll, als Liste von `SolutionExpression` Prädikaten, welche das Ergebnis der Übersetzung der Subexpresions sind. Die `function()` Funktion erstellt zunächst einen neuen Funktionsaufruf, welcher als Parameter die Ergebnisvariablen der übergebenen Parameter Expressions bekommt. Diese werden unter Verwendung des Mappingsyntax des `nemo_terms!` Makros (Unterabschnitt 2.2.4) extrahiert. Dieser Syntax wird auch danach in der Nemo Template Language Regel verwendet. Es ist anzumerken, dass das `nemo_terms!` Makro Duplikate Variablen automatisch entfernt, jedoch die Reihenfolge ansonsten beibehält, wodurch der Funktionsaufruf `call` immer noch als Letztes in der `rsult` Position des `SolutionExpression` Prädikats ist. Die `depend_vars()` Funktion der `SoluitonExpression` klasse gibt bei gegebenem Prädikat alle Variablen, die nicht die `result_var` sind zurück. Somit hat ein Funktionsaufruf alle Abhängigkeiten, die in einer der Subexpressions vorkommen. Wie bereits in anderen Expression-Übersetzungen besteht der *Regel-Körper* ↗ aus den Prädikaten für die Parameter und den Variablen-Bindings welche die Parameter limitieren, für welche die Funktion berechnet wird. Die Nemo Template Language Syntax erlaubt es optional nach dem zweiten Semikolon dem erzeugen Prädikat einen dynamischen Default-Namen zu geben. Wodurch das zurückgegebene Prädikat nicht den Namen „solution“, sondern den Namen der Nemo Funktion hat. Die korrekte Anzahl der Parameter muss nicht getestet werden, da dies bereist von spargebra validiert wird.

Der Funktionsaufruf `CONCAT(?a, ?x)` in SPARQL kann beispielsweise wie folgt übersetzt werden:

```
concat(?a, ?x, CONCAT(?RESULT, ?RESULT_1))
:- var(?a, ?RESULT), var_1(?x, ?RESULT_1), bgp(?a, ?x) .
```

Dabei sind `var` und `var_1` die Resultate der Übersetzung der Subexpressions für die Variablen `?a` bzw. `?x` und `bgp` *repräsentiert* ↗ die aktuellen Variablen-Bindings (wie dies bereits in Abschnitt 3.4 erläutert wurde).

Hier die Liste der direkt übersetzten Funktionen:

Categorie	Namen
General-purpose	DATATYPE, STR
Strings	STRLEN, UCASE, LCASE, CONCAT, LANG, REGEX
String tests	STRSTARTS, STRENDS, CONTAINS, STRBEFORE, STRAFTER
numbers	ABS, ROUND, CEIL, FLOOR
tests	isIri, isNumeric, isBlank

Es folgen noch zusätzliche Kommentare:

- Der SPARQL 1.1 Standard hat konkrete Regeln für die Verwendung von Plain-String, xsd:string und Language-Taged-String in einer konkreten Situation. Diese werden in Nemo grundsätzlich nicht berücksichtigt. Dies stellt keine größere nicht-Konformität dar, da das verhallten in SPARQL 1.2 bereits stark vereinfacht wurde. Ein größere nicht-Konformität stellt jedoch mangelnde Implementierung für Funktionen auf Language-tagged-Strings dar. So ergibt beispielsweise `CONCAT(ä"@de, "b"@de)` einen Error in Nemo.
- Die REGEX Funktion in Nemo verwendet eine Rust-Bibliothek, welche sich auf Effizienz konzentriert und deutlich weniger Features unterstützt als der in SPARQL verwendete Regex-Syntax. Außerdem werden keine Regex-Flags unterstützt.
- Die Ergebnisse einiger String-Funktionen hängen von der Definition eines characters ab. Die Länge des „technologist emoji" hat beispielsweise die Länge 1 in Nemo, in Blazegraph die Länge 5 und in Virtuoso die Länge 3. Der anzuwendende Standard ist dabei ISO/IEC 10646 (laut dem in SPARQL referenzierten xpath-Standard) und die korrekte Antwort 3: ein Person-Emoji, ein Zero-Width-Joiner und ein Laptop-Emoji. (Diese Arbeit wurde nicht von einem Unicode Experte Korrektur gelesen)
- Die Funktionen UCASE und LCASE sind vertauscht in der verwendeten Version von Nemo.
- `LANG()` ergibt keinen leeren String für Plain-Strings
- `ROUND()` rundet in die falsche Richtung für negative Werte, welche auf „.5" enden.
- `isBlank()` ist die `isNull()` Funktion in Nemo

Abgeleitete Funktionen

Einige SPARQL Funktionen werden unterstützt, obwohl sie nicht direkt in Nemo zur Verfügung stehen:

- Die `BNode()` Funktion Variante ohne Parameter erzeugt einen neuen BNode. Diese wird mittels einer existentiellen Regel ohne *Regel-Körper* ↗ übersetzt. Dies ist unterschiedlich von einem Fakt und wird von der Nemo Template Language unterstützt. Fakten in Nemo können keine existentiell quantifizierten Variablen enthalten. Im SPARQL Standard wird nicht offensichtlich definiert, wie oft diese Funktion evaluiert wird. Eine einmalige Evaluierung ist konsistent mit anderen SPARQL Implementierungen, z.B. Blazegraph.

- Die `SUBSTR()` Funktion wird mittels der `SUBSTR()` und `SUBSTRING()` Funktionen, je nachdem ob 2 bzw. drei Parameter übergeben wurden, übersetzt. Beide Funktionen sind in Nemo für die die Edge-Cases nicht Standardkonform. Zur Implementierung wird die `function()` Methode (Unterunterabschnitt 3.4.13) mit dem jeweils korrekten `func` Parameter aufgerufen.
- Die Funktion `isLiteral()` wird durch `NOT(OR(isIri(?x), isNull(?x)))` für gegebenen Parameter `?x` implementiert. Jeder Funktionsaufruf wird dabei einzeln mit der `function()` Methode (Unterunterabschnitt 3.4.13) übersetzt.
- Einige Datumsfunktionen werden mittels anderer Nemo Funktionen übersetzt. Siehe Unterunterabschnitt 3.4.13 dafür

Datums Funktionen

Einige Date-Time-Expressions in SPARQL dienen dazu, eine Komponente eines Date-Time-Literals zu extrahieren. Keine dieser Funktionen werden in der verwendeten Version von Nemo direkt unterstützt. Da jedoch der xsd:dateTime Type ein recht festen lexical space hat, sind einige davon recht leicht mithilfe anderer Nemo Funktionen zu übersetzen.

Der lexical space von xsd:dateTime ist im Standard definiert als

,-'? yyyy ,-' mm ,-' dd ,T' hh ,:' mm ,:' ss (,.' s+)? (zzzzzz)?

Danach kann man einen Date-Time-Wert als ein `T` in der Mitte, wobei Zahlen vor dem `T` mittels - getrennt und nach dem `T` mittels : getrennt sind, sehen. Außerdem gibt es noch zusätzlich die Zeitzone, deren Format separat beschrieben ist: `(('+' | '-') hh ':' mm) | 'Z'`. Es ist zu erkennen, dass die Zeitzone immer mit +, - oder `Z` beginnt.

Folgende Date-Time-Funktionen werden bei der Übersetzung unterstützt:

Name	is_time	offset
Year	false	2
Month	false	1
Day	false	0
Hours	true	0
Minutes	true	1
Secondes	true	2

Dabei gibt `is_time` an, ob sich der Wert vor oder hinter dem `T` befindet und `offset` ist der Index beginnend bei 0 neben `T`.

Basierend auf `is_time` kann mittels `STRBEFORE()` oder `STRAFTER()` der korrekte Teil extrahiert werden. Um das *n*-te Element im String zu finden wird *n* – 1 mal die `STRAFTER()` Funktion mit dem korrekten Separator angewendet und danach die `STRBEFORE()` Funktion mit dem gleichen Separator, um den restlichen String zu entfernen. Die `STRBEFORE()` Funktion wird nur angewendet, wenn es sich nicht um das letzte Element (`offset` = 2) handelt. Da für `is_time = false` der Index, um negative Daten zu unterstützen, von hinten beginnt, wird vor und nach dem Finden des *n*-ten Elements die `STRREV()` Funktion angewandt, um den String zu drehen. Zum Schluss wird der Zahl-String mit der `INT()` Funktion zu einem Integer konvertiert. Eine Ausnahme stellt die `SECONDS()` Funktion dar. Hier wird das Ergebnis zu `DOUBLE()` konvertiert und die Zeitzone entfernt, in dem alles nach +, - oder `Z` entfernt wird. Um alles nach einem bestimmten Symbol zu entfernen, kann mittels `CONCAT()` das

entsprechende Symbol angehängt werden und anschließend mittels `STRBEFORE()` sich auf den String bis zum ersten Auftauchen des entsprechenden Symbols beschränkt werden.

Die tatsächliche Übersetzung findet in der `date_function()` Funktion statt, diese umfasst 40 Zeilen inklusive einem Match-Statement, welches aus der übergebenen spargebra Funktion die `is_time` und `offset` Parameter bestimmt. Die Implementierung verwendet das `nemo_call!` Makro (Unterabschnitt 2.2.4) umfangreich.

Hier, als Beispiel, eine Übersetzung der `Month` Funktion:

```
extract_date(
    ?a,
    INT(
        STRREV(
            STRBEFORE(
                STRAFTER(
                    STRREV(
                        STRBEFORE(
                            STR(?date),
                            "T"
                        )
                    ),
                    "-"
                ),
                "-"
            )
        )
    )
)
:- var(?a, ?date), DATATYPE(?date) = <http://www.w3.org/2001/XMLSchema#dateTime> .
```

`var` ist das Prädikat der Subexpression. Es ist hier nicht nötig, die Variablen-Bindings noch einmal extra im *Regel-Körper*↗ zu haben, da die Datefunktionen nur von einer Subexpression, dem Datum, abhängen. Es ist nötig den Datentyp zu prüfen, da sonst die Funktion auch auf einigen speziell formatierten String funktionieren würde.

Einziger bekannter Unterschied der Implementierung zum SPARQL Standard ist der Rückgabetyp der `SECONDS()` Funktion, welcher im Standard xsd:decimal ist, jedoch bei der Implementierung absichtlich xsd:double, da Nemo xsd:decimal nicht umfangreich unterstützt.

Nicht unterstützte Funktionen

Einige Funktionen, die der SPARQL Standard fordert, werden nicht von der Übersetzung unterstützt:

Kategorie	Funktionen	Kommentar
String	LangMatches, Replace	Wären unter Umständen nichttrivial in Nemo zu implementieren
Erzeugung	Iri, StrLang, StrDt, BNode mit Argumenten	sind fundamental unmöglich, mit bestehenden Funktionen zu implementieren

Kategorie	Funktionen	Kommentar
Random	Rand, Uuid, StrUuid	Innerhalb einer Query Übersetzung wäre globales State-Management für Rand theoretisch mit bestehenden Mitteln möglich
Datum	Timezone, Tz, Now	für Tz wäre Implementierung mit bestehenden Funktionen möglich
Hashing	Md5, Sha1, Sha256, Sha384, Sha512	Wäre sicher leicht in Nemo zu implementieren
Sonstiges	EncodeForUri	Wäre sicher leicht in Nemo zu implementieren

Bist auf `LangMatches()` und `Replace()` wären alle Funktionen sicherlich sehr einfach in Nemo zu implementieren und auch `Replace` wäre kein Problem, wenn man sich mit dem reduzieren regex feature set wie bei der bestehenden Implementierung der `REGEX()` Funktion zufriedengibt.

Mit den Funktionen aus der Kategorie „Erzeugung“ ließe sich theoretisch der komplette SPARQL Standard in Nemo abbilden (da Nemo Turing-Complete ist und jede Operation auf Strings durchführen kann). Ohne diese Funktionen ist es jedoch bei dem vorgegebenen Ausgabeformat eines Nemo Prädikats welches das SPARQL *Query-Ergebnis↗ repräsentiert↗* unmöglich den Kompletten SPARQL Standard in Nemo zu übersetzen.

SPARQL Erweiterung für weitere von Nemo unterstützte Funktionen

Nemo unterstützt einige Funktionen, die nicht direkt Teil des SPARQL Standards sind. Diese müssten nicht in dieser Arbeit berücksichtigt werden. Es ist dennoch in der Praxis hilfreich einige davon zu unterstützen und eine entsprechende Implementierung ist sehr leicht, mit der bestehenden `function()` Methode aus Unterunterabschnitt 3.4.13 möglich. SPARQL erlaubt dafür Extensions durch Custom-Funktionen welche durch eine IRI gegeben werden. Folgende Custom-Funktionen werden bei der Übersetzung unterstützt:

Funktion	Nemo Funktion
math:sqrt()	SQRT
math:sin()	SIN
math:cos()	COS
math:tan()	TAN
math:pow()	POW
fn:sum()	SUM
fn:min()	MIN
fn:max()	MAX

Präfix math: http://www.w3.org/2005/xpath-functions/math#
Präfix fn: https://www.w3.org/2005/xpath-functions#

Für die restlichen Funktionen wurde keine offensichtliche IRI gefunden. Insbesondere ist `math:log()`, welche den natürlichen Logarithmus berechnet, nicht äquivalent zur ‘LOG()‘ Funktion in Nemo.

3.5 Property Paths

In SPARQL kann eine Einschränkung von einem „Start-Knoten“ und einem „End-Knoten“ (Knoten eines Graphs definiert in Abschnitt 1.3) durch eine Property Path Expression gegeben werden. Für jedes Ergebnis der Query muss der Pfad zwischen Start- und End-Knoten die Property Path Expression erfüllen. Ein Pfad ist eine Sequenz von Kanten, wobei Objekt-Position der vorhergehenden Kante mit der Subjekt-Position der darauffolgenden Kante übereinstimmen muss. Die Property Path Expression definiert eine Einschränkung der Prädikate in entlang der Sequenz, sehr ähnlich zu regulären Ausdrücken. Im Gegensatz zu vielen Programmiersprachen gibt es jedoch in dieser Art von regulären Ausdrücken in SPARQL keine komplexeren Features wie Look-Ahead. Dafür gibt es spezielle Features für Pfade in einem Graphen, wie beispielsweise das „überqueren“ einer Kante von Object zu Subject Position statt wie sonst von Subject zu Object Position.

In Datalog lässt sich das Ergebnis einer Property Path (Teil-)Expression durch ein Prädikat darstellen, welches die Menge aller Knotenpaare aus Start- und End-Knoten *repräsentiert* ↗ welche die Property Path Expression erfüllen. Da die resultierenden Prädikate immer zweistellig sind, kann auf die Definition von eigenem Prädikattypen der Nemo Template Language verzichtet werden. Es werden einfache Prädikate vom Typ `SolutionSet` verwendet. Wobei die erste Position der Start-Knoten und die zweite Position der End-Knoten ist.

Die Implementierung geschieht in der Methode

```
fn translate_path(
    &mut self, start: Binding, path: &PropertyPathExpression, end: Binding
) -> Result<SolutionSet, TranslateError>
```

in der `QueryTransloat` Klasse. Dabei ist `start` und `end` der Start bzw. Endpunkt des Pfades vom, in der Nemo Template Language definierten, Typ `Binding`, welcher sowohl Konstanten als auch Variablen darstellen kann. `path` ist die zu übersetzende `PropertyPathExpression` aus der Transformation zu SPARQL-Algebra mittels der spargebra Bibliothek. Das Ergebnis ist ein zweistelliges Prädikat vom Typ `SolutionSet`. Die Implementierung besteht aus einer Fallunterscheidung nach dem Typ der `PropertyPathExpression`. Hat diese Subexpressions werden diese zunächst rekursiv mit der `translate_path` Funktion übersetzt.

3.5.1 OneOrMore

Ein Feature in Property Path Expressions ist der `OneOrMore` Pfad dargestellt durch ein Suffix `+`. Beispielsweise kann folgende Query genutzt werden, um alle Subklassen der Klasse `ex:cat` anzufragen inklusive Subklassen von Subklassen:

```
SELECT ?type
WHERE {
    ex:cat ex:has_subclass+ ?type .
}
```

Die Klasse `ex:cat` selbst ist dabei nicht im Ergebnis, was den Unterschied zu einem `ZeroOrMore` Pfad ausmacht.

In Nemo lässt sich das verhallten durch eine rekursive Regel übersetzen. Das Prädikat `inner` *repräsentiert* ↗ im Beispiel das (wie in Abschnitt 3.5 definierte) Ergebnis der inneren Property Path Expression:

```
recursive(ex:cat, ?next) :- inner(ex:cat, ?next) .
recursive(ex:cat, ?next) :- recursive(ex:cat, ?mid), inner(?mid, ?next) .
```

Da der Anfang des Pfades im Beispiel eine bekannte Konstante ist, ist der erste Parameter die konstante `ex:cat`. Dennoch muss der innere Pfad auch für andere Startknoten berechnet werden.

Der Start- und End-Knoten können jeweils eine Variable oder Konstante sein. Diese 4 Möglichkeiten werden einzeln im SPARQL Standard betrachtet. Die Implementierung im Beispiel würde ähnlich funktionieren, wenn statt der Konstanten `ex:cat` der Anfang des Pfades eine Variable wäre. Damit die Implementierung auch für zwei Konstanten funktioniert, wird in der tatsächlichen Implementierung noch eine weitere Regel verwendet:

```
nemo_def!(one_or_more_path(start, end) :- recursive(start, end); SolutionSet);
```

Dabei ist `start` und `end` die Start- bzw. End-Konstante. Für den Fall, dass der End-Knoten eine Konstante und der Startknoten keine Konstante ist, gibt es eine zweite, dazu symmetrische, Implementierung, bei der in allen Regeln die erste und zweite Position der Prädikate vertauscht sind.

3.5.2 Pfade der Länge Null

Ein Pfad der Länge null bedeutet, dass der Pfad im selben Knoten endet, wie er anfängt. Im SPARQL Standard werden unterschiedliche Fälle betrachtet, je nachdem, ob Anfang und Ende Konstanten oder Variablen sind. Die Implementierung findet in der `zero_path_extend` Methode statt. Diese nimmt den Start, das Ende und das Prädikat `path`, welches bisher das Ergebnis der Übersetzung der Property Path Expression ist, als Parameter und fügt Regeln hinzu, damit das Prädikat auch Pfade der Länge null berücksichtigt.

Der Einsatz von Berechnung mit Optional in Rust ermöglicht es in der Implementierung, die beiden Fälle, in denen genau eins der beiden Pfadenden eine Konstante ist, gleichzeitig abzudecken. Es wird ein einfacher Fakt `nemo_add!(path(c, c));` hinzugefügt, wobei `c` die Konstante am Ende des Pfades ist.

Im Falle, dass der Pfad in einer Variablen beginnt und endet, kann zu jedem Knoten ein Pfad der Länge null, welcher bei diesem beginnt und endet, gesehen werden. Im SPARQL Standard wird dafür die Funktion `nodes(G) = { n | n is an RDF term that is used as a subject or object Tieffffffof a triple of G}` definiert. Bemerkenswerte ist dabei, dass nur Knoten, die als Subjekt oder Objekt vorkommen, beachtet werden. In der Nemo Template Language lässt sich dies wie folgt abbilden:

```
nemo_add!(path(?s, ?s) :- input_graph(?s, ?p, ?o));
nemo_add!(path(?o, ?o) :- input_graph(?s, ?p, ?o));
```

Der letzte Fall tritt ein, wenn Start und Ende beide Konstanten sind. Dieser Fall kann wie folgt implementiert werden, wobei `start` und `ende` die entsprechenden Konstanten sind:

```
nemo_def!(start_c(start));
nemo_def!(end_c(end));
nemo_add!(path(?c, ?c) :- start_c(?c), end_c(?c));
```

Diese Implementierung führt den Vergleich auf Gleichheit durch Verwendung der gleichen Variable in Nemo durch. Dies schafft Konsistenz, da die gleiche vorm von Gleichheit auch in der Übersetzung von Basic Graph Patterns verwendet wird und in SPARQL manche Property Path expressions equivalent zu einem Basic Graph Pattern sind und diese direkt von der spargebra Bibliothek zu einem Basic Graph Pattern übersetzt werden.

Hierzu ein kleiner Ausblick in die Edge-Cases von SPARQL. Es ist unklar im SPARQL Standard welche vorm von Gleichheit in diesem Spezialfall zu verwenden ist. Es wird ohne weiteren Kommentar der mathematische Ausdruck „X = Y“ verwendet. In der Praxis ist interessant, in welchen Fällen ein literal überhaupt der Anfang und Ende eines Pfades sein kann, denn Literale sind nicht in Subjektposition erlaubt in RDF. Hier ist ein Beispiel, wobei die Query auf einem leeren Graph ausgeführt wird:

```
SELECT DISTINCT (1 as ?out)
WHERE {1.0 (^<https://example.org/backward> / <https://example.org/forward>)? 1}
```

Empirisch ergibt die Query in Virtuoso einen Fehler, und in rdflib und Blazegraph sind `1.0` und `1` als unterschiedlich angesehen. Allerdings sind `"01"^^xsd:integer` und `"1"^^xsd:integer` unterschiedlich in rdflib und gleich in blazegraph. Das bedeutet, die Implementierung in Nemo ist in dieser Hinsicht Blazegraph ähnlich. Weitere untersuchen lassen sich BNodes. Der gleiche BNode `_:1` wird bei der Implementierung in Nemo als gleich angesehen, nicht aber für diesen speziellen fall in rdflib und blazegraph und Virtuoso wirft einen Fehler „BNode not allowed in subject of transitive path“. Noch interessanter wird es, wenn ein BNode mit gleicher ID in einem Basic-Graph-Pattern und einer Property-Path-Expression vorkommt. An sich ist der Scope der BNodes durch die Join Operation der SPARQL-Algebra, in diesem Fall für Property Path Expression und Basic Graph Pattern unterschiedlich, jedoch nicht bei property path expressions, welche äquivalent zu einem Basic Graph Pattern sind. Die spargebra Bibliothek fängt diesen sonderfall ab und wirft einen hilfreichen Fehler. Diesen Fehler kann man jedoch umgehen, in dem man einen anonymen BNode verwendet: `s:a s:b [ s:x* s:y ]`. Dieser Pattern enthält in der SPARQL-Algebra zwei unterschiedliche anonyme BNodes - einen für den Basic Graph Pattern und einen für die Property Path Expression.

Pfade der Länge null werden in der Implementierung der `ZeroOrMore` Expression und der `ZeroOrOne` Expression verwendet. Bei der `ZeroOrOne` Expression wird die innere Subexpression übersetzt und das Ergebnis um die Pfade der Länge null erweitert. Bei der `ZeroOrMore` Expression wird zunächst der innere Pfad mit Variablen als Start und Ende übersetzt, danach die Rekursion mittels der in Unterabschnitt 3.5.1 beschriebenen Methode durchgeführt und zuletzt die Pfade der Länge null hinzugefügt.

3.5.3 Negated Property Set

Negated Property Set ist eine Property Path Expression, welche einen Pfad aus einer Kante einzelnen matcht. Dabei ist die Property dieser Kannte keine aus einer gegebenen Liste von nicht erlauben Properties.

Die Übersetzung geschieht in der `translate_negated_property_set()` Methode. Diese Methode enthält folgende wesentliche Nemo Template Language Definition:

```
nemo_def!(
    negated_property_set(start, end)
    :- input_graph(start, ?property, end), {filters}
```

```
    ; SolutionSet
);
```

Dabei wurde `start` und `end` als Parameter übergeben und kann sowohl eine Variable als auch eine Konstante sein. Alle Einschränkungen für die `?property` Variable sind in den `filters` enthalten. Für jedes Element der gegebenen Liste an nicht erlauben Properties gibt es eine ungleich, `!=`, Bedingung in Nemo. Diese Bedingungen werden in einem For-Loop in Rust mit dem `nemo_atoms!` Makro (Unterabschnitt 2.2.4) aggregiert.

Bemerkenswerte ist, dass in diesem Fall keine Diskussion über die korrekte Variante von Ungleichheit durchgeführt werden muss, da in Prädikate-Position keine Literale erlaubt sind in RDF.

SPARQL erlaubt auch Syntax, bei dem manche properties in einem negated property set invertiert sind. Diese sind jedoch laut Standard äquivalent zu einer Kombination aus anderen Property Path features und die entsprechende Transformation wird bereits beim Übersetzen in SPARQL Algebra von der spargebra Bibliothek durchgeführt, sodass sie nicht weiter berücksichtigt werden müssen.

3.5.4 Weitere Property Paths

Die bisher noch nicht vorgestellten Property Path Expression Features sind leicht implementierbar und werden direkt in der `translate_path` Methode implementiert. `start` und `end` werden als Parameter übergeben und können jeweils eine Variable oder konstante sein.

- `NamedNode` - Ein Pfad über eine einfache Property: `path_property(start, end) :- input_graph(start, property, end)`
- `Reverse` - Der innere Property Path von Ende zu Start: `path_reverse(start, end) :- inner(end, start)` (Start und Ende müssen auch beim Übersetzen des inneren Property Paths vertauscht werden)
- `Sequence` - `path_sequence(start, end) :- first(start, middle), second(middle, end)` (`middle` ist eine neue Variable)
- `Alternative` - `first` Pfad oder `second` Pfad: `path_alternative(start, end) :- first(start, end)` und `path_alternative(start, end) :- second(start, end)`

Falls Start oder Ende eine Konstante sind, wird diese Information, wenn möglich, zu den Subexpressions propagiert. Dadurch wird das *Ergebnis*↗ der Prädikate kleiner und die Ausführung effizienter. Außerdem vereinfacht die spargebra Bibliothek einige Property Path Expressions, die sich zu einem Basic Graph Pattern umwandeln lassen. Dies führt zu einer deutlichen Effizienzsteigerung, da keine extra Prädikate für die Subexpressions verwendet werden und stattdessen Nemos Multiway-Join-Algorithmus zusätzliche Bedingugen aus einem zugehörigen Basic Graph Pattern direkt mit beachten kann.

3.6 Ergebnis-Typ-Transformationen

Es ist möglich die Ereignistypen `SolutionSet`, `SolutionMultiSet` und `SolutionSequence` (Siehe Abschnitt 2.3) ineinander zu transformieren. Dies ermöglicht es in vielen Fällen, Features nicht für alle Ergebnistypen zu implementieren und die verbleibenden Typen durch Konvertierung zu unterstützen. Es können die Typen verwendet werden, welche eine Natürliche Nemo Implementierung erlauben. Zwar muss in einigen Fällen Information „Frei

erfunden“ werden, beispielsweise muss bei der Konvertierung von `SolutionMultiSet` nach `SolutionSequence` eine Reihenfolge hinzugefügt werden, jedoch werden die entsprechenden Funktionen nur eingesetzt, wenn die Konvertierung Sinn macht, z.B. die Reihenfolge keine Rolle spielt.

3.6.1 Sequenz aus einem Set

```
fn get_sequence(&mut self, inner: &SolutionSet) -> SolutionSequence
```

Für die Konvertierung zu einer Sequenz muss für jedes Tupel ein beliebiger Index neu generiert werden. Dabei müssen die Indices eine lückenlose Abfolge von natürlichen Zahlen sein, welche bei null beginnt. Außerdem ist die Effizienz mitunter relevant, da fast jede Query ohne `DISTINCT` einen Basic Graph Pattern, bei dem die Reihenfolge keine Rolle spielt, in eine Sequenz umwandeln muss. (Die Konvertierung von Multiset zu Sequenz verwendet auch die `get_sequence()` Funktion zur Implementierung)

Es besteht ein Zusammenhang zwischen der Konvertierung zu einer Sequenz, bei der eine Beliebige Reihenfolge vergeben werden muss, und der Sortierung, bei der eine bestimmte Reihenfolge vergeben werden muss. Es könnte beispielsweise mittels einer existentiellen Regel für jedes Tupel ein BNode generiert werden und dann nach den BNodes sortiert werden. Dennoch kann nicht ein beliebiger Sortieralgorithmus verwendet werden. Beispielsweise benötigt die in Unterabschnitt 3.7.12 vorgestellte Bitonic-Sort Implementierung bereits eine Reihenfolge der Eingabe. Das ebenfalls dort vorgestellte Radix-Sortierverfahren würde hingegen funktionieren. Allerdings beeinflusst aufgrund der bereits oben erwähnten häufigen Verwendung der Implementierung eine lange Implementierung die Länge von vielen Query-Übersetzungen, was ein Arbeiten erschwert. Deswegen wurde sich für das ebenfalls in Unterabschnitt 3.7.12 vorgestellte „Hacker-Mans-Sort“ Verfahren entschieden. Der größte Nachteil, dass dieses Sortierverfahren keine SPARQL konforme Sortierreihenfolge erzeugt, spielt hier keine Rolle und die Vorteile der kurzen Implementierung und Ausführungszeit sind sehr erwünscht. Allerdings ist es abhängig von der spezifischen derzeitigen Nemo-Rule-Engine Implementierung, insbesondere davon, dass BNodes einer existentiellen Regel durch eine fortlaufende ID benannt werden. Hier eine Beispielübersetzung; da die Query fast keine anderen Features beinhaltet, ist hier die komplette Übersetzung dargestellt (Kommentare manuell eingefügt):

SPARQL-Query:

```
prefix ex: <https://example.org/>

SELECT ?a WHERE {
    SELECT DISTINCT ?a
    WHERE {?a ex:p ex:o}
}
```

Nemo-Programm:

```
% Innere Query
bgp(?a) :- input_graph(?a, <https://example.org/p>, <https://example.org/o>) .
projection_1(?a) :- bgp(?a) .

% Für jedes Tuple im Ergebniss ein neuer BNode
sequence_proto(?a, !bnode_for_id) :- projection_1(?a) .
```

```
% Extrahieren der Zahl aus dem BNode Namen
sequence_shifted(?a, INT(STR(?bnode_var))) :- sequence_proto(?a, ?bnode_var) .

% Beginn der Indicees auf 0 setzen
sequence_start(#min(?RESULT)) :- sequence_shifted(?a, ?RESULT) .
sequence(?id - ?min, ?a) :- sequence_shifted(?a, ?id), sequence_start(?min) .

% SELECT ?a implementierung
projection(?INDEX, ?a) :- sequence(?INDEX, ?a) .
@output projection .
```

Die verwendete Min-Aggregation hat den Nebeneffekt, dass die an dieser Stelle ein neues Stratum des Programms beginn. In der Praxis stellt dies sicher, dass z.B. nicht das `input_graph` Prädikat vom `projection` Prädikat abhängen kann und die existentielle Regel nur einmal bearbeitet wird. Würde sich beim Testen in Zukunft herausstellen, dass die Regel mehrfach aufgerufen wird und so Lücken in der Index Sequenz auftreten, müsste künstlich auch ein neues Stratum vor der existentiellen Regel eingeführt werden, es gibt aber derzeit keine bekannte Query bei der dies auftrat.

3.6.2 Sequenz aus einem Multiset

```
fn get_sequence_from_multi(&mut self, inner: &SolutionMultiSet) -> SolutionSequence
```

Die Konvertierung von einem Multiset zu einer Sequenz verwendet im Wesentlichen die `get_sequence()` Methode, die zur Konvertierung von einem Set zu einer Sequenz dient (Unterabschnitt 3.6.1). Zuvor werden jedoch zu jedem Tupel, je nach Multiplizität, weitere Tupel generiert. Diese Tupel unterscheiden sich durch eine zusätzliche Zahl. Konkret ist die zusätzliche Zahl zunächst die Multiplizität des Tupel verringert um 1 und für jede Multiplizität größer als 0 wird auch die nächst kleinere zusätzliche Zahl erzeugt:

```
let remaining = nemo_var!(remaining);
nemo_def!(
    pre_index(??vars, remaining.clone() - 1)
    :- inner(@count: remaining.clone(); ??vars)
    ; SolutionSet
);

nemo_add!(
    pre_index(??vars, remaining.clone() - 1)
    :- pre_index(??vars, remaining.clone()),
    {nemo_filter!("", remaining.clone(), " > 0")}
)
```

Nach dem `get_sequence()` Aufruf wird die zusätzliche Zahl wieder entfernt.

Wie in Unterabschnitt 3.7.12 gezeigt, sind solche zählenden Regeln in der Verwendeten Nemo Version quadratisch in ihrer Laufzeit. Besteht also die Multimenge aus einem einzigen Ergebnis der Multiplizität 100.000, ist diese Implementierung sehr langsam (etliche Minuten). In der Praxis ist dieser Fall zwar durchaus möglich, jedoch nicht so häufig. Eine effizientere Implementierung könnte anstatt die nächst kleinere zusätzliche Zahl auch direkt noch weitere

Zahlen erzeugt werden. Eine balancierte Baumstruktur ist jedoch nicht trivial und in den bisherigen Tests traten so hohe Multiplizitäten nicht auf.

3.6.3 Weitere Ergebnis-Typ-Transformationen

Weitere Ergebnistransformationen sind sehr einfach. Die Transformation einer Sequenz zu einem Multiset kann durch Zählen (Aggregation) der jeweils relevanten Indizes geschehen (`get_multi_from_sequence()` Methode entspricht der `ToMultiSet` Funktion im SPARQL Standard):

```
nemo_def!(
    as_multi(@count: %count(?i); ??vars)
    :- inner(@index: ?i; ??vars)
    ; SolutionMultiSet
);
```

Ein Multiset kann aus einem Set gebildet werden, indem die Multiplizität auf `1` gesetzt wird (`get_multi()` Methode):

```
nemo_def!(
    multi(@count: 1; ??vars) :- inner(??vars); SolutionMultiSet
);
```

Ein Set kann durch Weck lassen des `count` oder `index` geschehen. Für dies gibt es keine extra Funktion. Die Operation wird direkt an den Stellen, wo dies nötig ist, durchgeführt.

3.7 Graph Patterns

Graph-Patterns ist ein Typ von Operation in SPARQL-Algebra, welcher alle Operationen abdeckt, welche nicht Expressions, Property-Path-Expressions oder Query Forms sind. In der spargebra Bibliothek gibt es dazu den Enum-Typ `GraphPattern`. Die Übersetzung eines Graph-Patterns erfolgt durch ein `macht` Statement in Rust und individuellen Implementierungen für die einzelnen Operationen. Außer einzelne Operationen werden auch Sonderfälle für Kombinationen aus Operationen, wie eine Implementierung ohne Negation für `FILTER EXISTS` (Siehe Unterabschnitt 3.7.4), betrachtet.

Unterschiedliche Operationen können unterschiedliche Typen als Ergebnis haben. So ist beispielsweise bei einer `OrderBy` Operation die Reihenfolge der Ergebnisse relevant und bei einer Query ohne `DISTINCT` spielt die Häufigkeit eines Ergebnisses eine Rolle. Um dies abzubilden, gibt es drei Methoden in der `QueryTranslator` Klasse, die jeweils einen anderen Ergebnistyp haben:

- `fn translate_pattern(&mut self, pattern: &GraphPattern) -> Result<SolutionSet, TranslateError>`
- `fn translate_pattern_multi(&mut self, pattern: &GraphPattern) -> Result<SolutionMultiSet, TranslateError>`
- `fn translate_pattern_seq(&mut self, pattern: &GraphPattern) -> Result<SolutionSequence, TranslateError>`

Die Funktionen rufen sich gegenseitig rekursiv in den entsprechenden Fällen auf. Alle Information über die zu übersetzende Operation wird durch den `pattern` Parameter übergeben. Die Ergebnistypen `SolutionSet` (Ohne Zusatzinformation für jedes Ergebnis), `SolutionMultiSet` (Mit `count` für jedes Ergebnis) und `SolutionSequence` (Mit `index` für jedes Ergebnis) wurden bereits in Abschnitt 2.3 vorgestellt. Operation unabhängige Konvertierungen zwischen den Typen wurden in Abschnitt 3.6 vorgestellt.

3.7.1 Distinct und Reduced

Die `DISTINCT` Operation der SPARQL-Algebra entfernt alle Duplikate aus dem Ergebnis. Die `REDUCED` Operation stellt es der Implementierung frei einige Duplikate zu entfernen. Da die Darstellung von und Umgang mit `SolutionMultiSet` und `SolutionSequence` Overhead bedeutet, ist eine Query ohne Duplikate im Allgemeinen effizienter, weshalb `REDUCED` genau wie `DISTINCT` übersetzt wird.

Die Verwendung von `DISTINCT` entscheidet auch darüber, welchen Ergebnistyp die übersetzte Query am Ende produziert. Die Unterscheidung ist nur für `SELECT` queries relevant und findet in der `translate_query` Funktion (Abschnitt 3.8) statt. Ist die äußerste Operation `DISTINCT` oder ein `Slice` (`LIMIT` und `OFFSET`) von `DISTINCT` (`Slice` wird nach `DISTINCT` evaluiert), ist das Ergebnis ein `SolutionSet`. In allen anderen Fällen ist das Ergebnis eine `SolutionSequence`. Insbesondere resultieren Queries mit `REDUCED` in einer `SolutionSequence`. Der Typ `SolutionMultiSet` wird nur für interne Zwischenergebnisse verwendet.

Die Implementierung von `REDUCED` bzw. `DISTINCT` führt zu keinen eigenen Regeln in Nemo. Stattdessen wird die innere Query durch die Funktion `translate_pattern()` übersetzt, welche ein `SolutionSet` und damit keine Duplikate produziert. Für die Funktionen `translate_pattern_multi()` und `translate_pattern_seq()` wird das Ergebnis danach in den richtigen Typ konvertiert (Abschnitt 3.6).

Der SPARQL Standard schreibt vor, dass `DISTINCT` und `REDUCED` eine durch `ORDER BY` gegebene Ordnung erhallten müssen. Dies ist bei der Übersetzung nicht gegeben.

3.7.2 Basic Graph Patterns

Basic-Graph-Pattern, `BGP` in SPARQL-Algebra, ist ein grundlegendes SPARQL Feature zum Matchen von Graph teilen. Hier ein Beispiel:

```
prefix ex: <https://example.org/>

SELECT DISTINCT ?a {
    ?a ex:p ex:o .
    ?a ex:q ?x.
}
```

Die Query fragt alle Knoten, welche den Wert `ex:o` für die Property `ex:p` und einen beliebigen Wert für die property `ex:q` haben, an.

Die Übersetzung der Query in Nemo ist mit wenigen Regeln möglich und kann deswegen hier komplett abgebildet werden. Das Präfix `ex` wurde nachträglich eingefügt:

```
@prefix ex: <http://example.org/> .

bgp(?a, ?x) :- input_graph(?a, ex:p, ex:o), input_graph(?a, ex:q, ?x) .
projection(?a) :- bgp(?a, ?x) .
@output projection .
```

Das Prädikat `input_graph` muss separat mit den Tripeln des Eingabe-Graphen definiert werden. Es ist zu beobachten, dass jedes Triple im Basic-Graph-Pattern zu einem `input_graph` Prädikat im *Regel-Körper* ↗ des `bgp` Prädikats übersetzt wird. Indem der gesamte Basic-Graph-Pattern zu einer einzigen Regel übersetzt wird, kann Nemos effiziente Multiway-Join-Implementierung gut ausgenutzt werden. Außerdem wird der potenziell große Eingab-Graph auf diese Weise schnell zu einem potenziell deutlich kleineren Zwischenergebnis reduziert.

BNodes werden zunächst zu Nemo-Variablen übersetzt (In Abschnitt 3.3 beschriebenes vorgehen ermöglicht es aus den BNode IDs eindeutige Variablen zu bilden) und dann für `SolutionSet` ignoriert oder für `SolutionMultiSet` mit `#count` Aggregation als Multiplizität gezählt. Da die `#count` Aggregation in Nemo mindestens eine Variable als Parameter benötigt, gibt es einen extra betrachteten Sonderfall, falls der Basic Graph Pattern keine BNodes enthält, welcher die Multiplizität für alle Ergebnisse auf 1 setzt. Der SPARQL Standard definiert keine Reihenfolge für die Ergebnisse eines Basic-Graph-Patterns, deswegen wird für `SolutionSequence` die `SolutionMultiSet` Implementierung verwendet und das Ergebnis konvertiert (Abschnitt 3.6).

Die Implementierung der Übersetzung umfasst folgende Methoden in der `QueryTranslator` Klasse:

- `translate_term()` - Transformation der IRIs, Literals, BNodes und Variablen aus der spargebra Repräsentation in Nemo Template Language sowie Verwaltung von Listen der Variablen und BNode-Variablen
- `translate_triple()` - Transformation eines spargebra Tripels in ein Nemo Template Language `input_graph` Prädikat und weitere Verwaltung der BNodes und Variablen. Die Funktion verwendet für Subjekt- und Objekt-Position die `translate_term()` Funktion, die Prädikat-Position hat einen eigenen Typ in spargebra.
- `translate_bgp()` - BGP ohne Multiplizität.
- `translate_bgp_multi()` - BGP mit Multiplizität.

Im SPARQL Standard wird die Semantik eines Basic Graph Patterns durch ein „pattern instance mapping“ *P* definiert, welches direkt zu einem Mapping der Variablen im *Regel-Körper* ↗, unter Berücksichtigung der Tripel im Graph, welcher durch das Prädikat `input_graph` *repräsentiert* ↗ wird, korrespondiert. Ein Ergebnis des Basic Graph Patterns μ ist dann „the restriction of *P* to the query variables“, was direkt zum `bgp` Prädikat korrespondiert, wobei das Mapping von den Positionen (welche nach Unterabschnitt 1.5.1 Variablen verkörpern) zu einem Tupel aus dem *Ergebnis* ↗ stattfindet. Die Multiplizität, gegeben μ, ist definiert durch „number of distinct RDF instance mappings, σ, such that $P = \mu(\sigma)$“ (for some valid *P*). Es ist möglich, die `#count` Aggregation in Nemo durch einen ähnlichen Ausdruck zu definieren.

Eine interessante Beobachtung ist, dass durch diese Definition der Scope eines BNodes auf innerhalb eines Basic-Graph-Patterns beschränkt ist. Folgende Query enthält zwei unterschiedliche BNodes mit gleicher ID (das BIND stellt sicher, dass die beiden BGPs nicht als ein BGP betrachtet werden können):

```
SELECT ?c WHERE {{_:1 a ?c}{_:1 a ?d . BIND(?d as ?c)}}
```

In rdflib wird der Scope dennoch ausgedehnt. In der verwendeten spargebra Bibliothek wird davon ausgegangen, dass dies ungewollt ist und ein entsprechender Parsing Error erzeugt.

3.7.3 Join Patterns

Die Graph-Patterns `Join` und `LeftJoin`, die SPARQL-Algebra-Operation zu `OPTIONAL`, sowie die `Exists` Expression (Unterabschnitt 3.4.9) haben, trotz unterschieden im Detail, ähnliches verhalten und sind dadurch sinnvoll zusammen zu betrachten.

Grundsätzlich lässt sich ein Join leicht in Datalog umsetzen, da der *Regel-Körper* ↗ in Datalog einen Join beschreibt. Das Beispiel implementiert einen Join, anhand der jeweils ersten Position, von zwei Tupel Mengen, *repräsentiert* ↗ durch die predikate x und y:

```
z(?a, ?b, ?c) :- x(?a, ?b), y(?a, ?c) .
```

Diese Join-Übersetzung liefert jedoch keine korrekten Ergebnisse, falls eine Tupel in der ersten Position den *UNDEF-Marker* ↗ enthält. Nach der obigen Implementierung würde ein *UNDEF-Marker* ↗ in x nur mit einem *UNDEF-Marker* ↗ in y matchen. Das korrekte verhallten ist jedoch, dass ein nicht definierter Wert in x mit jedem beliebigen Wert in y matcht. Damit ein Wert zu mehreren Werten matchen kann, kann ein Mapping Prädikat eingeführt werden, welches die Werte, die zueinander matchen, in Relation setzt. Da so ein Prädikat weniger effizient ist, wird getrackt, welche Variable überhaupt den *UNDEF-Marker* ↗ annehmen kann (Siehe Abschnitt 2.2). Nur für solche Variablen wird ein Mapping Prädikat generiert. Hier ist ein Beispiel für einen Join wobei die erste Position von y den *UNDEF-Marker* ↗ annehmen kann:

```
map(?a, ?a, ?a) :- x(?a, ?b), y(?a, ?c) .
map(?a, ?a, UNDEF) :- x(?a, ?b) .
z(?a, ?b, ?c) :- map(?a, ?left, ?right), x(?left, ?b), y(?right, ?c) .
```

Die zweite und dritte Position des `map` Prädikats bilden zwei Werte, die auf der linken bzw. rechten Seite des Joins matchen können. Die erste Position ist der Wert, welcher das Ergebnis des Joins ist. Die erste Regel stellt sicher, dass zwei gleiche Werte matchen, wobei das Ergebnis der matchende Wert ist. Die zweite Regel stellt sicher, dass der *UNDEF-Marker* ↗ in der ersten Position von y mit jedem Wert in der ersten Position von x matcht, wobei das Ergebnis der Wert aus x ist.

Im allgemeineren Fall, wo auch die erste Position von x den *UNDEF-Marker* ↗ annehmen kann, würde eine dritte Regel ähnlich der 2. Regel hinzukommen, welche bewirkt, dass der *UNDEF-Marker* ↗ in x mit jedem Wert der ersten Position aus y matcht. Weiterhin benötigt man ein Prädikat ähnlich dem `map` Prädikat aus dem Beispiel für jede Position, die in beiden zu joinenden Teilergebnissen vorkommt und welche nicht definiert sein kann.

Betrachtet man die Speichernutzung durch ein `map`-Prädikat, kann man sagen, dass die erste Regel höchstens so viele Tupel wie das kleinste *Ergebnis* ↗ der zu joinenende Prädikate erzeugt. Die andern beiden Regeln erzeugen höchstens so viele Tupel wie das *Ergebnis* ↗ der zu joinendenden Prädikate zusammen. Somit *repräsentiert* ↗ das `map` Prädikat höchstens doppelt so viele Tupel wie die zu joinenden Prädikate zusammen. Auf diese Weise kann gezeigt werden, dass durch das `map`-Prädikat keine massiv größeren Zwischenergebnisse, wie beispielsweise ein Zwischenergebnis, welches quadratisch größer als die Eingabe des Joins ist, entstehen kann.

Die Multiplizität $card(\mu)$ eines Solution-Mapping μ (Unterabschnitt 1.5.1) im Join Ergebniss ist in SPARQL wie Folgt definiert (Vereinfachte Darstellung):

$$card(\mu) = \sum card(\mu_1) * card(\mu_2) \text{ for } (\mu_1, \mu_2) \text{ such that } \mu = merge(\mu_1, \mu_2)$$

Dabei ist μ_1 und μ_2 im jeweils linken bzw. rechten zu joinenden Prädikat und 'merge()' (könnte auch „instance-level Join" genannt werden) kombiniert zwei Solution-Mapping wenn dies Möglich ist[6]. Die Summe wird benötigt, da unterschiedliche Kombinationen zum selben Ergebnis führen können. Z.B. führen die Tupel (1, 2) und (1, *UNDEF*) auf der linken Seite mit dem Tupel (1, 2) auf der rechten Seite beide zum Tupel (1, 2) im *Ergebnis*↗ des Joins. Die Formel aus dem SPARQL Standard kann direkt in Nemo übersetzt werden. Dafür hier das relevante Fragment in Nemo Template Language:

```
nemo_def!(
    join(
        @?count: %sum(cl.clone() * cr.clone(), extra_vars)
        ; result_terms
    ) :-
        {maps},
        left(@?count: cl; left_bindings),
        right(@?count: cr; right_bindings)
    ; T
);
```

Hier die Erklärung der verwendeten Variablen:

- `extra_vars`:
  alle Hilfsvariablen, im Beispiel oben `?left` und `?right`
- `result_terms`:
  Die Variablen im Ergebnisprädikat des Joins.
- `maps`:
  Die `map` Prädikate, bereits an die korrekten Variablen *gebunden*↗
- `left` / `right`:
  repräsentieren↗ die zu joinenden Zwischenergebnisse
- `left_bindings` / `right_bindings`:
  Die korrekten Variablen für die jeweiligen Positionen der `left` und `right` Prädikate
- `cl` / `cr`:
  Neue Variablen (Variablen in Expressions müssen vorher erstellt werden)
- `T`:
  Generischer Rust-Typ Parameter vom Typ `TypedPredicate` (Unterabschnitt 2.2.2)

Durch `@?`-Syntax wird der `count` Parameter nur gesetzt, wenn der verwendete Prädikat-Typ diesen unterstützt (Unterabschnitt 2.2.2). Dadurch ist die Implementierung generisch für `SolutionSet` und `SolutionMultiSet` einsetzbar. Siehe auch Unterabschnitt 2.2.4 für weitere Erklärungen von Features der Nemo Template Language.

Die Implementierung der Operatoren `Exists`, `Join` und `LeftJoin` umfasst in der Praxis fast 200 Zeilen und soll hier kurz vorgestellt werden. Alle Implementierungen verwenden die gemeinsame Funktion `unbound_join_map()` welche, bei gegebenem linken und rechten Prädikat und einer konkreten SPARQL Variable, ein einzelnes `map` Prädikat mit den dazugehörigen Nemo Regeln konstruiert. Die Funktion stellt fest, welche der Variablen am Join für

[6] https://www.w3.org/TR/sparql11-query/#defn_algCompatibleMapping

Regel 1 teilnehmen können und ob für eine der Seiten keine Regel nach der Form 2 bzw. 3 generiert werden muss. Die Funktion `unbound_join_map()` wird von zwei unterschiedlichen Funktionen verwendet, welche die Berechnung der korrekten Variablen-Bindings, zur Verwendung für die Prädikate in der finalen Regel bei der Berechnung des Joins, durchführen. Dabei bindet die Funktion `unbound_combinations()` die eigentliche SPARQL-Variable an erster Stelle des `map` Prädikats und die Funktion `unbound_combinations_left_focus()` an zweiter Stelle, sodass bei Verwendung dieser Variante immer das linke Prädikat das Ergebnis bestimmt. Außerdem hat die `unbound_combinations_left_focus()` eine einfachere Logik beim Generieren der Bindings für das linke Prädikat, da hier, weil das Ergebnis direkt vom linken Prädikat kommt, die SPARQL-Variablen, welche den Positionen ohnehin zugeordnet sind, direkt übernommen werden können.

Hier ein Vergleich der Implementierungen mit Join `translate_positive_exists()`, `translate_minus_g()` und `translate_join_multi_g()`:

- Funktion welche die Übersetzung durchführt
  - Join: `translate_join_multi_g()`
  - Minus: `translate_minus_g()`
  - Exists: `translate_positive_exists()`
- Verwendete Funktion zum generieren der Bindings
  - Join: `unbound_combinations()`
  - Minus: `unbound_combinations_left_focus()`
  - Exists: `unbound_combinations_left_focus()`
- Linker Join Input
  - Join: linke Eingabe
  - Minus: linke Eingabe
  - Exists: Variablen-Bindings zum evaluieren der Expression (siehe Abschnitt 3.4)
- Rechter Join Input
  - Join: rechte Eingabe
  - Minus: rechte Eingabe
  - Exists: exists-pattern Prädikat
- Egebnis Prädikat Typ
  - Join: `SolutionSet` / `SolutionMultiSet` (Unterstützt beide Typen durch generischen Parameter, deswegen auch das `_g` Suffix im Funktionsnamen)
  - Minus: `SolutionSet` / `SolutionMultiSet` (Ebenfalls generisch)
  - Exists: `SolutionExpression`
- Zusätzliche Bindings
  - Join: `@count` wie oben beschreiben
  - Minus: `@count` von linker Eingabe
  - Exists: `@result: true`

Die Implementierungen für Exists und Minus beinhalten noch weitere schritte außer einem Join, diese sind beschreiben in Unterabschnitt 3.4.9 für Exists und Unterabschnitt 3.7.7 für Minus.

Die Implementierung für `LeftJoin` (SPARQL-Algebra-Operation zu `OPTIONAL`) ist etwas komplexer und daher nicht direkt in der Tabelle oben aufgenommen. Es gibt zwei grundsätzlich unterschiedliche Implementierungen. Die erste Implementierung wird verwendet für Ergebnisse mit Multiplizität in der Funktion `translate_left_join_multi()`. Sie führt die `LeftJoin` Operation auf die Operationen `Join`, `Filter`, `Union` und `Exists` zurück. Im SPARQL Standard ist `LeftJoin` als

$$\texttt{LeftJoin}(\Omega_1, \Omega_2, expr) = \texttt{Filter}(expr, \texttt{Join}(\Omega_1, \Omega_2)) \cup \texttt{Diff}(\Omega_1, \Omega_2, expr)$$

definiert. Ω_1 ist das Testergebnis des eigentlichen Patern, Ω_2 ist dies Teilergebnis im `OPTIONAL`-Block und *expr* ist eine zusätzliche Filter-Expression. `Diff` ist wiederum definiert als Ω_1 ohne Ω_2 und ohne die Ergebnisse für die `expr` den Effective-Boolean-Value von `false` hat. Dabei können in `expr` sowohl Variablen aus Ω_1 als auch aus Ω_2 verwendet werden. Die Multiplizität bei `Diff` die Multiplizität aus Ω_1. `Diff` lässt sich durch eine Kombination aus `Filter` und `Exists` implementieren. Hier ist die Struktur der Operationen für `LeftJoin`(Ω_1, Ω_2, *expr*):

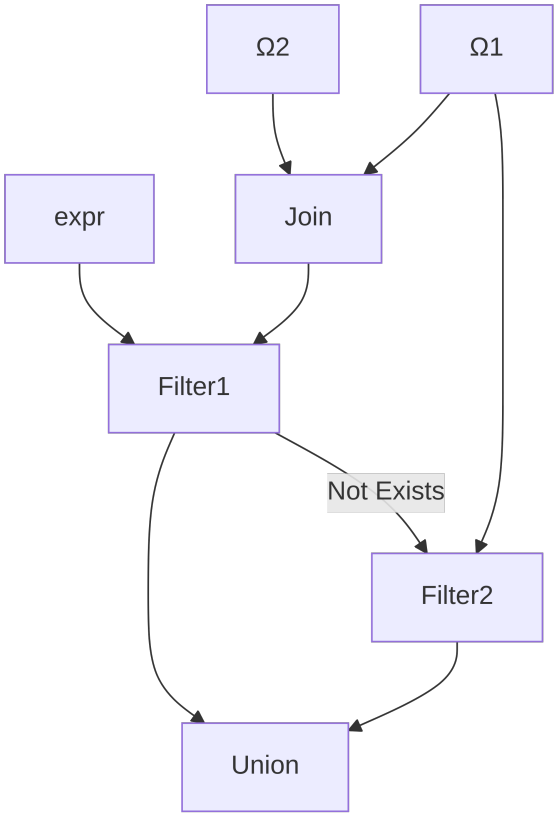

Abbildung 3.2: Datenfluss der Übersetzung von `LeftJoin(Ω1, Ω2, expr)`

Entscheidend ist dabei, dass das Ergebnis von Filter2 nur Variablen aus Ω_1 enthält, während das Ergebnis von Filter1 Variablen aus Ω_1 und Ω_2 bindet. Damit produziert die Union Operation *UNDEF-Marker* ↗ für alle Variablen aus Ω_2 für Tupel aus Ω_1 für die Filter1 kein passendes Ergebnis liefert.

Die Korrektheit der Diff Implementierung folgt aus doppelter Negation der im Standard definierten Bedingung und Propagierung der inneren Negation nach den erweiterten demoranschen Gesetzen. Aus der Defintion im SPARQL standard dafür das ein Ergebnis μ aus Ω_1 auch ein Ergebnis von `Diff` ist

$$\forall \mu' \in \Omega_2 : \text{either } \mu \text{ and } \mu' \text{ are not compatible or } \mu \text{ and } \mu' \text{ are compatible and } expr(merge(\mu, \mu')) \text{ has an effective boolean value of } \texttt{false}$$

wird dadurch

$$\text{not } \exists \mu' \in \Omega_2 : \mu \text{ and } \mu' \text{ are compatible and } expr(merge(\mu, \mu')) \text{ has an effective boolean value of } \texttt{true}$$

Wobei das „not ∃“ zu einem `FILTER NOT EXISTS`, die *expr*, welche true sein muss, zu einem Filter und das *merge* der kompatiblen solution-mappings direkt zu einem Join Korrespondiert. Für einen formaleren Beweis siehe Anhang——-TODO. Dort wird auch erklärt, warum das Gegenteil von „has an effecitve boolean value of `false`“ in diesem Fall „has an effective boolean value of `true`“ ist, obwohl es korrekt „has an effective boolean value of `true` or evaluates to an error“ wäre.

Die Multiplizität ist korrekt, da Filter und Diff diese direkt aus Ω_1 übernehmen.

Der filter not exists Teil der Implementierung beinhaltet stratifizierte Negation, was die Frage aufwirft, ob auch eine Implementierung ohne stratifizierte Negation und damit ein stratifiziertes Nemo-Programm bei welchem der Eingabe-Graph vom Ergebnis der SPARQL-Query abhängt, möglich wäre (Abschnitt 1.1). Grundsätzlich kann dies zu Widersprüchen führen, wenn beispielsweise der *UNDEF-Marker*↗ aus dem `Optional` dazu führt, dass nun ein Wert für das entscheidende Prädikat existiert, was ursprünglich zum *UNDEF-Marker*↗ geführt hat. Hier eine Implementierung ohne stratifizierter Negation, sie führt Negation durch Regeln mit existentiell quantifizierten Variablen (Unterabschnitt 1.5.2) durch:

Query:

```
prefix ex: <https://example.org/>

SELECT DISTINCT ?a ?b {
    ?a ex:p ex:o .
    OPTIONAL { ?a ex:q ?b . }
}
```

Übersetzung: (manuell mit Kommentaren versehen und Formatiert):

```
% computation before left join
bgp(?a)
    :- input_graph(?a, <https://example.org/p>, <https://example.org/o>) .

bgp_1(?a, ?b)
    :- input_graph(?a, <https://example.org/q>, ?b) .

% regular join
```

```
join(?a, ?b) :- bgp(?a), bgp_1(?a, ?b) .

% create dummy dependency
dummy_dependency(arity_zero) .
dummy_dependency(arity_zero) :- join(?a, ?b) .

% add bnodes to fill missing values
left_join_proto(?a, ?b, defined)
    :- join(?a, ?b), dummy_dependency(arity_zero) .

left_join_proto(?a, !b, !CONST)
    :- bgp(?a), dummy_dependency(arity_zero) .

% translate bnodes to unbound
dummy(arity_zero) .
undefined_val(UNDEF) :- dummy(arity_zero) .
left_join(?a, ?b) :- left_join_proto(?a, ?b, defined) .
left_join(?a, ?unbound)
    :- undefined_val(?unbound),
        left_join_proto(?a, ?b, ?CONST),
        ?CONST != defined .

% computation after left join
projection(?a, ?b) :- left_join(?a, ?b) .
@output projection .
```

Der reguläre Join muss dabei auch mit dem *UNDEF-Marker*↗ umgehen. Außerdem wird auch der Fall eines LeftJoin mit Filterexpression berücksichtigt. Beides ist jedoch im vorliegenden Beispiel nicht vorhanden. Die `dummy_dependency` bewirkt, dass beide `left_join_proto` Regeln in jeder Query Durchlauf Iteration in der richtigen Reihenfolge „ausgeführt" werden. Ohne `dummy_dependency` versagt die existential Regelevaluierungsheuristik bei größeren Queries. Wie oben bereits erwähnt, kann diese Übersetzung ohne stratifizierter Negation nicht für alle Fälle korrekte Ergebnisse liefern. In der Praxis ist jedoch ein korrektes verhallten, insbesondere im Standard-Fall ohne Abhängigkeit des Eingabe-Graph vom Ergebnis der Query, zu beobachten. Siehe auch Unterabschnitt 3.7.6 dazu.

3.7.4 Filter

Die Filter Operation in SPARQL behält nur Solution-Mappings (Unterabschnitt 1.5.1) für die eine gegebene Expression zu einem Ergebnis mit Effective-Boolean-Value `true` evaluiert. Sie ist durch eine einzige Nemo-Template-Language-Regel realisierbar:

```
nemo_def!(
    filter(@?count: ?c; ??expr_vars, ??other_vars)
    :- inner(@?count: ?c; ??expr_vars, ??other_vars),
       expression(??expr_vars; @result: true)
    ; T
);
```

Dabei *repräsentiert*↗ `inner` das zu filternde Zwischenergebnis und `expression` ist ein Prädikat vom Typ `SolutionExpression` wobei das Ergebnis vom Typ Boolean sein muss. Das

`??expr_vars` VarSet (VarSets beschrieben in Unterabschnitt 2.2.1) steht für alle Variablen, von denen die Expression abhängt.

Wie im gezeigten Nemo Template Language Fragment zu sehen wird die Multiplizität, falls benötigt, direkt vom inneren Zwischenergebnis übernommen. Die Implementierung wird in 4 Fällen eingesetzt:

Algebra-Operationen	Typ des Ergebnisprädikats	Übersetzung der Expression durch
Filter Exists	`SolutionSet`	`translate_positive_exists()`
Filter	`SolutionSet`	`translate_bool_expression()`
Filter Exists	`SolutionMultiSet`	`translate_positive_exists()`
Filter	`SolutionMultiSet`	`translate_bool_expression()`

Der Sonderfall für Exists ist in Unterabschnitt 3.4.9 näher beschrieben. Falls die Expression nicht ohnehin ein Ergebnis vom Typ Boolean garantiert (siehe Unterabschnitt 3.4.3), berechnet die `translate_bool_expression()` Funktion den Effective-Boolean-Value. Ergebnisse vom Typ `SolutionSequence` müssen erst als `SolutionMultiSet` gefiltert und dann konvertiert werden (Siehe Abschnitt 3.6) da ein bloßes filtern keinen fortlaufenden Index garantiert.

3.7.5 Union

Im Kern ist eine Union Operation ein „oder". Ein Tupel ist im *Ergebnis*↗, wenn es entweder im linken *oder* im rechten Teilergebnis ist. Dies kann durch zwei Regeln abgebildet werden:

Query:

```
prefix ex: <https://example.org/>
SELECT DISTINCT ?a {
    {?a ex:p ex:o1}
    UNION
    {?a ex:p ex:o2}
}
```

Übersetzung: (Predikate ohne effect manuell entfernt, und Umformatiert)

```
bgp(?a) :- input_graph(
    ?a, <https://example.org/p>, <https://example.org/o1>
) .
bgp_1(?a) :- input_graph(
    ?a, <https://example.org/p>, <https://example.org/o2>
) .
union(?a) :- bgp(?a) .
union(?a) :- bgp_1(?a) .
@output union .
```

Die tatsächliche Implementierung umfasst trotz der sehr einfachen Grundlage 60 Zeilen, was durch den Umgang mit Variablen, welche für ein Ergebnis nur auf einer der beiden Seiten gebunden sind, und drei eigene Implementierungen für `SolutionSet`, `SolutionMultiSet` und `SolutionSequence` zustande kommt.

Variablen, die nur auf einer Seite gebunden werden, müssen im Ergebnis zum *UNDEF-Marker*↗ gemappt werden. Dafür gibt es die `map_unbound()` Funktion. Diese berechnet Bindings für ein `target` Prädikat, sodass alle SPARQL-Variablen, die nicht in einem `source` Prädikat vorkommen zum *UNDEF-Marker*↗. Da dieser erst zur Laufzeit bekannt sein kann, werden die entsprechenden Positionen zu einer Variable gemappt, welche dann durch das einstellige `unbound` Prädikat, welches nur den *UNDEF-Marker*↗ im *Ergebnis*↗ hat[7], *gebunden*↗ wird. Außer mehrfacher Verwendung der `map_unbound()` Funktion in allen Union Implementierungen (jeweils für linke und rechte Teillösung) wird diese auch bei der Übersetzung von `left_join` und `project` (konkret beim Zusammenführen von Solution-Mappings, die durch das Project gleich werden) verwendet.

Grundlegend erweitern alle Implementierungen die linke und rechte Seite durch *UNDEF-Marker*↗ um sie aneinander anzupassen und führen dann beide Ergebnisse durch zwei Regeln, wie im initialen Beispiel oben, zusammen. Abhängig vom Ergebnistyp gibt es extra Logik. Die `SolutionMultiSet` Implementierung könnte dabei aus der `SolutionSequence` Implementierung abgeleitet werden, da sie jedoch in der `left_join_multi()` Implementierung (Unterabschnitt 3.7.3) verwendet wird, ist eine eigene effizientere Implementierung nützlich.

Funktion	Ergebnistyp	Extra Logik
`translate_union()`	`SolutionSet`	Keine
`translate_union_multi()`	`SolutionMultiSet`	Multiplizität wird addiert für Ergebnisse in beiden Seiten; insgesamt drei Regeln für Ergebnis links, rechts und in beiden
`translate_union_seq()`	`SolutionSequence`	Länge des linken Ergebnisses wird auf den Index des rechten Ergebnis addiert; extra Nemo Regel zur Berechnung der Größe

Nur die `SolutionSet` Implementierung kommt dabei ohne Features aus, die einem stratifizierten Programm entgegenwirken. Die `SolutionMultiSet` Implementierung benötigt stratifizierte Negation, um festzustellen, dass ein Ergebnis in einer Seite aber nicht der anderen Seite vorkommt und die `SolutionSequence` Implementierung verwendet Aggregation, um die Größe des linken Teilergebnisses zu bestimmen.

3.7.6 Extend

In SPARQL kann das Ergebnis einer Expression mittels `BIND` einer Variablen zugewiesen werden. In SPARQL-Algebra ist dafür die `Extend` Operation verantwortlich. Der SPARQL Standard lässt das verhallten absichtlich undefiniert, wenn die Variable bereits an einen anderen Wert gebunden ist. In der Praxis gibt es keine einheitlich verwendete Lösung. Virtuoso ignoriert z.B. das zusäzliche Binding, Blazegraph matcht bestehende Variable mit dem Binding und rdflib überschreibt das bestehende Binding. Spargebra, und somit auch die SPARQL zu Nemo Übersetzung, wirft einen Fehler in diesem Fall.

Eine Übersetzung von Extend könnte prinzipiell wie folgendes Beispiel funktionieren:

```
extend(?a, ?b, ?c, ?d) :- input(?a, ?b, ?c), expr(?b, ?c, ?d) .
```

[7] Formal korrekt ein ein-Tupel aus dem *UNDEF-Marker*↗

Dabei *repräsentiert*↗ `input` das erweiterte Ergebnis, `expr` das Ergebnis der Expression, wobei die Expression von den Variablen `?b` und `?c` abhängt, und `extend` das Ergebnis der Extend Operation. Das Ergebnis der Expression `?d` wird im Ergebnis der Extend Operation mit aufgenommen.

Falls die Expression in einem Fehler resultiert, ist die neue Variable nicht definiert, also im Kontext der Übersetzung der *UNDEF-Marker*↗. Da ein Error durch fehlendes Tupel für diese Expression-Inputs dargestellt wird, liese sich dies durch stratifizierte Negation implementieren. Da jedoch existentielle Regeln in Nemo ein sehr ähnliches Feature unterstützen, bei dem nur neue BNodes erzeugt werden, wenn nicht bereits ein Ergebnis existiert, stellt sich die Frage, ob eine Implementierung ohne stratifizierte Negation mittels existentieller Regeln möglich wäre. Dies ist im Allgemeinen nicht möglich, da bei zyklischer Abhängigkeit des Eingabe-Graphn vom Query-Ergebnis ein *UNDEF-Marker*↗ im Ergebnis zu einem neuen Wert, welcher den Fehler verhindert und damit den *UNDEF-Marker*↗ im Nachhinein inkorrekt machen würde, führen kann. Dieser Fall ist jedoch unmöglich mit monotoner Logik lösbar. Da jedoch der SPARQL Standard nicht für solche zyklischen Abhängigkeiten definiert ist, kann in solchen Fällen, die über den SPARQL Standard hinaus gehen, ein zusätzlicher *UNDEF-Marker*↗ im Ergebnis trotzdem zu praktisch nützlichen Ergebnissen führen.

Hier ist die Übersetzung einer einfachen Query mit `BIND`:

Query:

```
prefix ex: <https://example.org/>
SELECT DISTINCT ?a {
    ex:x ex:p ?a
    BIND (?a as ?b)
}
```

Übersetzung: (Kommentare nachträglich zur Erklärung der Implementierung eingeführt)

```
% computation before extend
bgp(?a) :- input_graph(
    <https://example.org/x>, <https://example.org/p>, ?a
) .
dummy(arity_zero) .
undefined_val(UNDEF) :- dummy(arity_zero) .
var(?a, ?a) :- bgp(?a), ~undefined_val(?a) .

% create dummy dependency
dummy_dependency(arity_zero) .
dummy_dependency(arity_zero) :- var(?a, ?RESULT) .

% add bnodes to fill missing values
proto_extend(?a, ?b, no_error)
:- bgp(?a), var(?a, ?b), dummy_dependency(arity_zero) .

proto_extend(?a, !no_result, !error)
:- bgp(?a), dummy_dependency(arity_zero) .

% translate bnodes to unbound
extend(?a, ?b) :- proto_extend(?a, ?b, no_error) .
extend(?a, ?RESULT)
:- proto_extend(?a, ?b, ?CONST),
```

```
    undefined_val(?RESULT),
    ?CONST != no_error .

% computation after left join
projection(?a) :- extend(?a, ?b) .
@output projection .
```

Es ist anzumerken, dass die Negation von `undefined_val` keinen Einfluss auf die Stratifizierbarkeit des Programms hat, da das `undefined_val` Prädikat nicht vom Eingabegraphen abhängt und immer zuerst berechnet werden kann. Empirisch kann festgestellt werden, dass das `dummy_dependency` Prädikat die Korrektheit bei größeren Queries, ohne dass der Eingabe-Graph vom Ergebnis abhängt, bewirkt. Bei kleinen Queries ist Nemos Heuristik zur Evaluation von existentiellen Regeln auch ohne `dummy_dependency` ausreichend. Siehe auch Unterabschnitt 3.7.3 für eine ähnliche Ersetzung von stratifizierter Negation durch existentielle Regeln.

3.7.7 Minus

`MINUS` ist neben `FILTER NOT EXISTS` das andere Modell Negation in SPARQL-Queries zu formulieren. Der SPARQL Standard definiert das Ergebnis der *Minus*(*A*, *B*) Operation als alle Solution-Mappings in *A* für die kein Solution Mapping in *B* existiert, was kompatibel ist und mindestens eine Variable gemeinsam hat. Kompatibel ist dabei eine häufig im SPARQL Standard verwendete vorm von „Gleichheit“ (nicht immer transitiv) von Solution Mappings (siehe Unterabschnitt 1.5.1) bei der nicht definierte Variablen mit allen Werte matchen. Die extra-Bedingung, dass gemeinsame Variablen nötig sind, ist einer der Unterschiede, wenn Negation mit `MINUS` anstatt mit `FILTER NOT EXISTS` ausgedrückt wird. (Der andere Unterschied ist, dass bei `FILTER NOT EXISTS` Variablen aus dem zu Filterenden Zwischenergebnis im Scope sind. Was jedoch wie in dieser Arbeit nicht standardkonform umgesetzt wurde, siehe Unterabschnitt 3.4.9)

Die Übersetzung berechnet die Operation in zwei Schritten. Zunächst werden die zu entfernenden Tupels durch ein Prädikat `to_remove` berechnet und dann werden diese Tupels mittels stratifizierter Negation entfernt und daraus das `minus` Prädikat berechnet. Der erste Schritt ergibt alle Tupels aus *A* für die es ein kompatibles Tupel aus *B* gibt und erfolgt grundsätzlich ähnlich zur `Exists` Implementierung (Unterabschnitt 3.7.3). Es wird zunächst für alle Variablen, die nicht definiert sein können, ein Mapping Prädikat gebildet, welches jeden relevanten Wert aus dem anderen Prädikat mit dem *UNDEF-Marker* ↗ in Relation setzt und jeden anderen Wert mit sich selbst in Relation setzt. Dafür wird die bereits in Unterabschnitt 3.7.3 beschriebene Funktion `unbound_combinations_left_focus()` wieder verwendet. Die Minus Implementierung unterscheidet sich zur Exists Implementierung durch das Prüfen der Extrabedinung, dass Tupel nur matchen, wenn sie mindestens eine gemeinsame Variable haben. Die jeweiligen Fälle sind durch folgende Auflistung gegeben:

- Die gegebenen Prädikate haben keine gemeinsame Variable
  - **Detection:** Einfacher Test während der Übersetzung
  - **Vorgehen:** `Minus` hat keinen Effekt, zurückgeben der Linken Eingabe
- Für eine Variable ist Definiertheit in beiden Prädikaten bekannt
  - **Detection:** Information wird gesammelt während Konstruktion der Mappingprädikate

  - **Vorgehen:** extra-Bedingung wird aus Nemo-Programm entfernt (ist immer erfüllt, muss nicht geprüft werden)

- Es gibt eine Variable für die beide Prädikate keinen *UNDEF-Marker*↗ haben
  - **Detection:** Evaluierung zur Laufzeit in Nemo (siehe unten), extra-Bedingung ist erfüllt
  - **Vorgehen:** Tupel ist im *Ergebnis*↗ des `to_remove` Prädikats
- Für alle Variablen hat eines der Prädikate einen *UNDEF-Marker*↗
  - **Detection:** Evaluierung zur Laufzeit in Nemo (siehe unten), extra-Bedingung ist nicht erfüllt
  - **Vorgehen:** Tupel nicht im *Ergebnis*↗ des `to_remove` Prädikats

Die extra-Bediung ist eine oder Verknüpfung über alle Variablen und für jede Variable wird geprüft, ob diese für das eine Prädikat und für das andere Prädikat definiert ist. Der *UNDEF-Marker*↗ kann theoretisch jeder Wert sein. Ein vergleich zum *UNDEF-Marker*↗ mittels = in Nemo wäre möglich, liefert jedoch keinen Boolean, welchen man für die `AND()` bzw. `OR()` Funktion in Nemo nutzen könnte. Eine Umsetzung mit mehreren Regeln wäre denkbar, wenn auch weniger effizient. Es ist jedoch auch eine Implementierung als ein Constraint (siehe Unterabschnitt 1.5.2) in einer Regel möglich. Der Vergleich kann dabei mit `ABS(COMPARE(fullStr(?val), fullStr(?undef)))` durchgeführt werden, was `0` ergibt, wenn beide unterschiedlich, also der Wert definiert und `1` wenn der Wert nicht definiert ist. `1` und `0` lassen sich jedoch nicht so leicht in einen Boolean Wert für die Verwendung von `AND` bzw. `OR` konvertieren. Stattdessen kann `*` für `AND` und + für `OR`, in diesem Zusammenhang, genutzt werden. So wird das Ergebnis des Vergleiches für eine Variable für beide Prädikate multipliziert und die Ergebnisse für alle Variablen wiederum addiert. Sind alle Variablen in mindestens einem Prädikat nicht definiert, ist das Ergebnis `0`. Ist das Ergebnis größer als `0` ist mindestens eine Variable in beiden Prädikaten definiert, was die gesuchte Extrabedingung darstellt. Genau genommen muss der Vergleich nur für Positionen, von denen nicht bereits durch Tracken bekannt ist, dass sie definiert sind, durchgeführt werden. Kann also eine Variable nur für ein Prädikat den *UNDEF-Marker*↗ annehmen, kann der Vergleich direkt dazuaddiert werden und es muss für diese Variable keine Multiplikation erfolgen. (Siehe Abschnitt 2.2 für das Tracken von Eigenschaften)

Das `to_remove` Prädikat ist vom Typ `SolutionSet` und das `minus` Prädikat hat einen generischen Typ (entweder `SolutionSet` oder `SolutionMultiSet`) für `SolutionMutliSet` wird mittels `@?`-Syntax (Unterabschnitt 2.2.2) die Multiplizität vom ersten Operand übernommen. `SolutionSequence` wird nicht direkt unterstützt und muss aus dem `SolutionMutliSet` konvertiert werden.

3.7.8 Values

Bei `Values` wird das Ergebnis der SPARQL Operation direkt als Eingabe vom Autor der Query angegeben. Bei der Übersetzung wird ein gegebener Wert zunächst von spargebra in eine spargebra spezifisch Repräsentation transformiert und dann mittels der `translate_ground_term()` Funktion in Nemo Template Language Repräsentation übersetzt. Das eigentliche `values` Prädikat wird dann mit einer Regel für jedes Tupel erzeugt. Eine Regel ist notwendig statt einem Fakt, da der *UNDEF-Marker*↗ zur Übersetzungszeit nicht Bekannt sein muss und als Prädikat gegeben ist. Hier die Übersetzungsfunktion (Formatierung und Kommentare zur Erklärung der Implementierung nachträglich bearbeitet):

```
fn translate_values_seq(
    &mut self,
    variables: &Vec<Variable>,
    bindings: &Vec<Vec<Option<GroundTerm>>>
) -> SolutionSequence {
    // create new Predicate with correct position variables
    let values = SolutionSequence::create(
        "values",
        nemo_terms!(
            nemo_var!(index),
            variables => self.translate_var_func()
        ).vars()
    );

    let undef = nemo_var!(undef);
    let undef_pred = self.undefined_val.clone();
    for (i, binding) in bindings.iter().enumerate() {
        // convert the values to nemo template languge
        let terms: Vec<Binding> = binding.iter().map(
            |b| self.translate_ground_term(b, &undef)
        ).collect();

        // Add the tuple
        nemo_add!(values(@index: i; terms) :- undef_pred(undef));
    }
    values
}
```

Die Inputtypen `Variable` und `GroundTerm` sind in spargebra definiert. Die Funktion erzeugt ein Nemo Template Language Prädikat `values` vom Typ `SolutionSequence`. `SolutionMultiSet` Ergebnisse müssen durch Konvertierung erzeugt werden. Für `SolutionSet` gibt es eine eigene sehr ähnliche Funktion. Das `values` Prädikat hat an erster Stelle den Index und an den anderen Stellen die vom SPARQL-Query-Autor gegebenen Variablen. Diese werden mittels des Mapping-Syntax (siehe Unterabschnitt 2.2.4) des `nemo_terms!` Makros in Variablen der Nemo Template Language übersetzt. Die eigentliche Übersetzung passiert mittels der `translate_var_func()` Funktion, diese gibt eine Funktion zurück, welche die korrekte Variable in einem `VariableTranslator` Objekt des `QueryTranslator` (`self` Parameter) anfragt. Es kann keine neue Variable erzeugt werden, da unterschiedliche Variablen automatisch umbenannt werden, um Namenskollisionen zu verhindern (siehe Unterabschnitt 2.2.5).

Hier noch ein Beispiel:

Query:

```
SELECT ?a {
    VALUES (?a ?b) {(1 UNDEF) (2 3)}
}
```

Übersetzung: (explizite xsd:integer Datentypen aus Literalen nachträglich entfernt und Kommentare hinzugefügt)

```
% setup UNDEF Marker
dummy(arity_zero) .
```

```
undefined_val(UNDEF) :- dummy(arity_zero) .

% values pattern
values(0, 1, ?undef) :- undefined_val(?undef) .
values(1, 2, 3) :- undefined_val(?undef) .

% final projection
projection(?INDEX, ?a) :- values(?INDEX, ?a, ?b) .
@output projection .
```

3.7.9 Project

Für `Values` (Unterabschnitt 3.7.8) gab es eine separate Übersetzung für mit bzw. ohne Multiplizität, da die `@?`-Syntax nicht für die generische Implementierung ausreichte, denn es wurde ein explizites Prädikat erstellt und es muss bei den unterschiedlichen Typen unterschieden werden, wie viele Variablen bei der Erstellung angegeben werden. Die `Project` Implementierung ist leicht komplexer und unterstützt alle drei Typen `SolutionSet`, `SolutionMultiSet` und `SolutionSequence`. Dadurch lohnt sich eine generische Implementierung, obwohl auch hier das `projection` Prädikat manuell erstellt wird. Die Unterscheidung, ob eine zusätzliche initiale Variable bei der Erstellung übergeben werden muss, wird festgestellt, in dem die Länge aller Variablen mit der Länge der innern Variablen verglichen wird. Die Funktion `inner_vars()` ist für jeden der drei Typen definiert und gibt alle Variablen eines Prädikats abgesehen von den extra Variablen für Multiplizität oder Index zurück. Die eigentliche Übergabe passiert mittels des `nemo_terms!` Makros (Unterabschnitt 2.2.4), wobei auch wie bei `Values` die übergebenen zu projizierenden Variablen mittels der `translate_var_func()` Funktion gemappt werden.

An sich kann eine Projektion sehr einfach in einer Nemo Regel stattfinden, indem das `projection` Prädikat im *Regel-Kopf* ↗ nur die zu projizierenden, während das zu projizierende `inner` Prädikat im *Regel-Körper* ↗ alle Variablen gebunden hat. Es sind jedoch in SPARQL zwei Sonderfälle zu beachten. Zum einen sind dies projizierte Variablen, die nicht im zu projizierenden Zwischenergebnis vorkommen, und zum anderen sind dies Tupel, welche vor der Projektion unterschiedlich und nach der Projektion gleich sind.

Im SPARQL Standard ist die Projektion die Einschränkung der Variablen, für die jedes einzelne Solution Mapping (Unterabschnitt 1.5.1) definiert ist, ist ein Solution Mapping bereist vorher für eine Variable nicht definiert ist dieses Mapping nach der Projektion für diese Variable immer noch nicht definiert. Existiert eine Variable nicht im `inner` Prädikat, bedeutet dies, dass sie für keins der Solution Mappings definiert ist. Da sie auch nach der Projektion für kein Solution Mapping definiert ist, wäre ein Valide Ansicht jede Variable, die nicht im `inner` Prädikat vorkommt zu ignorieren, jedoch ist aus Sicht eines Anwenders der SPARQL zu Nemo Übersetzung wünschenswert, dass man leicht die Arität des resultierenden Nemo Prädikats anhand des Project (`SELECT`) Pattern der SPARQL Query ablesen kann. Deswegen wird die nirgends definierte Variable konstant auf den *`UNDEF`-Marker* ↗ gesetzt. Gibt es eine solche Variable wird diese, in einem Zwischenschritt, welcher ein `proto_projection` Prädikat ergibt, hinzugefügt.

Tupel, die durch Projektion gleich werden, sind nur bei `SolutionMultiSet` relevant. Bei `SolutionSequece` unterscheiden sich ohnehin alle Tupel im Index und bei `SolutionSet` werden die Gleichen Tupel automatisch zusammengefasst. Bei `SolutionMultiSet` können sich Tupel hingegen allein durch die Multiplizität unterscheiden. Diese Multiplizitäten müssen durch `sum` Aggregation zusammengefasst werden

Die wichtigste Zeile der Übersetzung kann hier gezeigt werden:

```
nemo_add!(
    projection(@?count: %sum(?c, ??other), @?index: ?i; ??projected)
    :- extended_inner(@?count: ?c, @?index: ?i; ??projected, ??other)
);
```

Es wird das `nemo_add!` Makro statt `nemo_def!` (Unterabschnitt 2.2.4) verwendet, da das `projection` Prädikat bereits zuvor mit den korrekten Variablen erstellt werden musste, somit wird das VarSet (Unterabschnitt 2.2.1) `??projected` automatisch an die korrekten zu projizierenden Variablen gebunden und `??other` steht für die verbleibenden Variablen. Das `extended_inner` Prädikat ist das zu projizierende Teilergebnis, welches, falls nötig, bereits um nicht im Teilergebnis vorkommende Variablen erweitert wurde. Es ist zu sehen, dass mittels des `@?`-Syntax (Unterabschnitt 2.2.2) sowohl Multiplizität mit `count` als auch Sortierung mittels `index` generisch unterstützt wird.

3.7.10 Slice

`Slice` ist die SPARQL-Algebra-Operation zu `LIMIT` und `OFFSET`. Dies lässt sich für Zwischenergebnisse welche durch ein Prädikat vom Typ `SolutionSequence` *repräsentiert*↗ werden durch eine Condition (Unterabschnitt 1.5.2) der Index-Variable in einer Nemo-Regel und eine Verschiebung der Indices im Ergebnis durch Subtraktion implementieren:

```
fn translate_slice_seq(
    &mut self,
    inner: &SolutionSequence,
    start: usize,
    length: Option<usize>
) -> SolutionSequence {
    let index = nemo_var!(index);
    let mut condition = nemo_condition!(index, " >= ", start);
    if let Some(l) = length {
        condition = nemo_atoms![
            condition,
            nemo_condition!(index, " < ", start + l)
        ]
    }
    nemo_def!(
        slice(@index: index.clone() - start; ??vars)
        :- inner(@index: index; ??vars), {condition}
        ; SolutionSequence
    );
    slice
}
```

Die anderen Typen können dadurch durch Konvertierung von bzw. nach `SolutionSequence` unterstützt werden (Abschnitt 3.6).

Es ist anzumerken, dass `LIMIT` in SPARQL häufig dazu verwendet wird, eine Query ohne Anspruch auf Vollständigkeit, effizienter zu machen, da die Evaluierung abbrechen kann, wenn bereits genügend Ergebnisse gefunden wurden. Dieses Prinzip wird hier nicht berücksichtigt, da grundsätzlich alle Zwischenergebnisse komplett berechnet werden. Dennoch

kann das Feature verwendet werden, z.B. wenn eine Subquery limitiert wird, sodass eine Äußere Query auf weniger Ergebnissen ausgeführt wird.

3.7.11 Aggregation

Aggregation ist in SPARQL recht komplex, mittels der Operationen `Group`, `ListEval`, `Aggregation`, `AggregateJoin` und `Flatten` sowie extra Operationen für die jeweiligen Aggregationsfunktionen `Count`, `Sum`, `Avg`, `Min`, `Max`, `GroupConcat` und `Sample`, definiert. Spargebra hat bereits ein vereinfachtes Model und übersetzt Aggregation zu einem Graph Pattern `Group`, welcher Aggregates vom Typ `AggregateExpression` haben kann. Zusätzlich bettet spargebra die `Group` Operation in eine `Extend` Operation (Unterabschnitt 3.7.6). Dadurch werden äußere Expressions, in denen das Ergebnis der Aggregation modifiziert wird, wie bei `#sum(...) + 1`, bereits inklusive Fehlerbehandlung ohne zusätzlichen Aufwand abgedeckt. Bei der Übersetzung wird eine SPARQL Aggregatione im Wesentlichen zu einer Nemo Aggregation übersetzt und die Details aus dem SPARQL Standard somit nicht einzeln adressiert.

Dennoch werden die grundlegenden Features, welche in SPARQL relevant sind, aber nicht direkt in Nemo unterstützt werden, adressiert:

- Unterstützung für mehrere Aggregationen (Nemo unterstützt nur eine Aggregation pro Regel)
- Unterstützung von Aggregation mit und ohne `DISTINCT` Keyword (Nemo verwendet, Set-Semantik nach der es grundsätzlich keine Duplikate geben kann)
- Unterstützung von komplexen Expressions (Bei der Übersetzung werden auch manche Expressions unterstützt, die nicht direkt in Nemo zulässig sind)

Die Übersetzung einer einzelnen Aggregation findet in der Funktion `translate_aggregation()` statt:

```
fn translate_aggregation(
    // QueryTranslator
    &mut self,
    // repräsentiert zu aggregierendes Teilergebniss
    inner: &dyn TypedPredicate,
    // SPARQL Variable für ergebniss der Aggregation
    variable: VarPtr,
    // zu aggregierende Expression
    expression: &Expression,
    // name der aggregationsfunktion in Nemo
    aggregation: &str,
    // Variablen nach denen grupiert werden soll
    group_vars: &Vec<VarPtr>,
    // ob aggregation in Nemo "distinct variables" nicht unterstützt
    idempotent: bool
) -> Result<SolutionSet, TranslateError>
```

Zunächst wird die spargebra `expression` wie bereits beschrieben (Abschnitt 3.4) mit der `translate_expression()` Methode zum `expr_solution` Prädikat übersetzt. Danach wird der Aufruf der Aggregationsfunktion mit korrekten Parametern `aggregation_call` (für nicht idempotente Funktionen werden alle nicht grupierten Variablen als „distinct variables“[8] in

[8] https://github.com/knowsys/nemo/wiki/Rule-language#distrinct-variables

Nemo angegeben) und das `aggregate` Prädikat mit allen `group_vars` und der `variable` für das Ergebnis der Aggregation generiert. Schließlich wird die eigentliche Regel erzeugt:

```
nemo_add!(
    aggregate(??group_vars, aggregation_call)
    :- inner(??group_vars, ??other), {&expr_solution}
);
```

Für `count(*)` gibt es eine eigene Funktion `translate_count_all()` welche, statt einer zu aggregierenden Expression, alle nicht gruppierten Variablen mittels einer `#count` Aggregation in Nemo aggregiert. Der Spezialfall, dass bei allen Variablen gruppiert wird, wird nicht extra beachtet und führt zu einem Fehler in Nemo, da eine Aggregation ohne Parameter in Nemo nicht erlaubt ist. Korrekt sollte das Ergebnis in diesem Fall 1 sein.

Alle Aggregationen werden in der Funktion `translate_group_by()` zusammengeführt:

```
collect_aggregations(?b, ?var_26, ?e0)
:- bgp(?a, ?b), sum_aggregate(?b, ?var_26), count_aggregate(?b, ?e0) .
```

Dies ist ein Ausschnitt der Übersetzung von volgender Query:

```
SELECT DISTINCT (SUM(DISTINCT ?a) as ?s) (COUNT(DISTINCT ?a) as ?c) ?b
WHERE { ?a ex:p ?b . }
GROUP BY ?b
```

Die Variablen für die Ergebnisse der Aggregationen sind von spargebra generierte Hilfsvariablen und werden in der nachfolgenden `extend` Operation verwendet. Die Namen wurden im Beispiel manuell gekürzt. Außerdem ist zu beobachten wie so eine Variable im Beispiel mit 2 beginnt und damit nicht valide in Nemo ist und mit einem Präfix `var_` valide gemacht wurde.

Die korrekten Aggregationen werden in einer Fallunterscheidung erzeugt:

Distinct	spargebra Expression	Nemo Aggregation	Übersetzungsfunktion	idempotent
Ja	Count mit `expr = None`	-	`translate_count_all()`	-
Ja	Count sonst	`#count`	`translate_aggregation()`	Nein
Ja	Sum	`#sum`	`translate_aggregation()`	Nein
Ja	Min	`#min`	`translate_aggregation()`	Ja
Ja	Max	`#max`	`translate_aggregation()`	Ja
Nein	Count mit `expr = None`	-	`translate_count_all()`	-
Nein	Count sonst	`#count`	`translate_aggregation()`	Nein
Nein	Sum	`#sum`	`translate_aggregation()`	Nein
Nein	Min	`#min`	`translate_aggregation()`	Ja
Nein	Max	`#max`	`translate_aggregation()`	Ja

Die Fälle mit `DISTINCT` verwenden dabei das `inner` Prädikat, welches mit der `translate_pattern()` Funktion erstellt wurde, als relevantes Zwischenergebnis und die Fälle ohne `DISTINCT` verwenden das `inner_multi` Prädikat, welches mit der `translate_pattern_multi()` Funktion erstellt wurde (siehe Abschnitt 3.7). In

der Tabelle wurde keine Angabe der Nemo Aggregation oder Idempotentheit für die `translate_count_all()` Funktion gemacht, da diese keine solchen Parameter erfordert.

Die im SPARQL Standard vorgesehenen Aggregationen `Avg`, `GroupConcat` und `Sample` werden nicht unterstützt, wobei zumindest `Avg` leicht aus `Sum` und `Count` implementierbar wäre.

Andere Ergebnisprädikat-Typen außer `SolutionSet` werden durch Konvertierung unterstützt. (Abschnitt 3.6)

3.7.12 Order By

Die Order By Operation ist mit Abstand die komplexeste Operation, wenn es um eine Übersetzung nach Nemo geht. Grundsätzlich stellt sich die Frage, wie überhaupt eine Reihenfolge in der Mengen-Semantik von Nemo dargestellt werden kann. Der verwendete Ansatz, wobei die erste Stelle eines Prädikats der Index des Ergebnisses angibt, wurde bereits in der gesamten Arbeit verwendet und insbesondere in Abschnitt 2.3 vorgestellt. Als Alternative wurde auch eine separate Nachfolgerrelation in Betracht gezogen. Diese lässt sich jedoch leicht aus der Index Repräsentation erstellen, da der Nachfolger Index einfach durch Addition von eins berechnet werden kann. Umgekehrt ließe sich der Index aus einer Nachfolgerelation zwar auch berechnen, ist jedoch in der Praxis erheblich langsamer, weswegen der andere Ansatz gewählt wurde.

Dieser Performance unterschied kann durch folgendes Programm demonstriert werden:

```
n(0) .
n(?i+1) :- n(?i), ?i + 1 < 100000 .
m(?i+1) :- n(?i), ?i + 1 < 100000 .
```

Die erste Regel benötigt dabei auf einem Testsystem ca. 13 min während die Zweite nur ca. 14 s benötigt, obwohl beide grundsätzlich sehr ähnlich sind. Die erste Regel ist jedoch signifikant langsamer, da sie durch die Rekursion für jeden neuen Wert neu aufgerufen werden muss. Führt man eine quadratische Regression auf gemessenen Zeiten in Abhängigkeit vom Maximalwert für ?i bei der ersten Regel durch, kann eine sehr gut passende quadratische Funktion gefunden werden, obwohl die Regel auf den ersten Blick linear aussieht. Wenn man den Eingabewerten einen fortlaufenden Index zuordnen will, z.B. beim Sortieren, muss sehr darauf geachtet werden, dies nicht wie in Regel eins zu tun. Zwar ist der Reasoningprozess in Nemo sequentiell, allerdings zeigt der Test, dass eine Regel, die einmal evaluiert wird und viele Ergebnisse produziert, erheblich effizienter als eine Regel, die häufig evaluiert wird und immer nur wenige neue Ergebnisse liefert, ist. Schwer parallelisierbare Algorithmen neigen dazu, Regeln zu beinhalten, die häufig aufgerufen werden müssen. Viele klassische Sortierverfahren sind nicht so gut parallelisierbar, was einer effizienten Implementierung in Nemo entgegenwirkt. Außerdem basieren viele Sortierverfahren darauf, dass ein beliebiger Wert wie z.B. der erste Wert in der liste als Pivoelement „genommen“ wird. Gibt es jedoch grundsätzlich keine Reihenfolge ist dies nicht umsetzbar, Nemo bietet keine Möglichkeit ein beliebiges Tupel eines Prädikats auszuwählen. Zwar kann mittels Aggregation z.B. der kleinste Wert „genommen“ werden. Dies verhindert jedoch Rekursion einer Regel, da sonst ein nicht stratifiziertes Programm entsteht und innerhalb eines Sortiervorgangs muss prinzipiell beliebig oft ein Wert „genommen“ werden, was Rekursion unvermeidbar macht.

Es wurden einige Algorithmen prototypisch implementiert. Einige gute Kandidaten werden hier vorgestellt und wurden auf zufällig generierten Zahlen getestet, die gemessenen Zeiten verändern sich in mehreren Messversuchen nur wenig und stellen die Zeit ohne generieren

und laden der Eingabedaten dar. Bei den Nemo-Programm Auszügen bildet das Prädikat `num` die zu sortierenden Zahlen bzw. das Prädikat `s` die zu sortierenden Strings.

Hacker Mans Sort

Hacker-Mans-Sort sortiert die Eingabe nicht selbst sondern verschafft sich Zugang zu der bereits sortierten Eingabe. In Nemo ist dies prinzipiell möglich da die Prädikate in Nemo bereits sortiert gespeichert werden. Es ist möglich diese sortierung sichtbar zu machen, da BNode namen in der Reihenvolge in der die Werte gespeichert sind sequentiell generiert werden. z.B. `_:5`, `_:6`, `_:7` ...

```
dummy(d) .
bnode(!x) :- dummy(d) .
proto_sort(!x, ?i, ?oi) :- num(?i, ?oi) .
shifted(INT(STR(?x)), ?i, ?oi) :- proto_sort(?x, ?i, ?oi) .
min_id(#min(?id)) :- shifted(?id, ?i, ?oi) .
sorted(?id - ?m, ?oi, ?i) :- min_id(?m), shifted(?id, ?i, ?oi) .
```

Auf 5 Mio. Zeilen dauerte das sortieren nur 11 Sekunden. Das Sortierverfahren ist jedoch schwer, mit unterschiedlichen Typen standardkonform umzusetzen und funktioniert beispielsweise nicht mit Strings, welche nach Nemos internen Indices und nicht alphabetisch sortiert werden.

Sort by Counting

Dies ist der einfachste Sortieralgorithmus in Nemo. Der Index eines Elements ergibt sich aus der Anzahl der Elemente, die kleiner sind.

```
sorted(#count(?other), ?s) :- s(?s), s(?other), COMPARE(?other, ?s) < 0 .
```

Das Verfahren lässt sich grundsätzlich auf allen Datentypen, für die eine Vergleichsoperation möglich ist, anwenden. Theoretisch wäre es sogar möglich für Nemo die `#count` Aggregation in konstanter Zeit durchzuführen, wenn die Werte bereits korrekt sortiert gespeichert sind. In der Praxis hat das Verfahren jedoch eine quadratische Laufzeit und ist dadurch sehr langsam für große Eingaben.

Bitonic Sort

Wie bereits erwähnt, bieten sich parallelisierbare Algorithmen für die Implementierung in Nemo an. Bitonic-merge-sort ist ein paralleler Sortieralgorithmus. Die hier vorgestellte Variante wurde in mehreren Iterationen optimiert. Die Kommentare erklären die Grobe Funktionsweise:

```
lut(-3, 1) .
lut(-2, 1) .
lut(-1, 1) .
lut(0, 0) .
lut(1, 0) .
lut(2, 0) .
lut(3, 0) .
```

```
% erste Position von s ist der fortlaufende initiale Index
len(#max(?i) + 1) :- s(?i, ?v) .

% shape(?t, ?w, ?s) mit
% ?t Zeitschritt,
% ?w Weite der blauen äußeren Box,
% ?s Größe der Braunen bzw. Roten Subbox
shape(0, 1, 1) .

shape(?t + 1, ?w * 2, ?w * 2)
:- shape(?t, ?w, 1), len(?l), ?w * 2 < ?l .

shape(?t + 1, ?w, ?s / 2)
:- shape(?t, ?w, ?s), ?s >= 2 .

% compare(?t, ?i1, ?i2)
% vergleichen der Werte am Index ?i1 und ?i2 zum Zeitschritt ?t
compare(?t, ?i, ?i + 2*?w - 1 - 2*REM(?i, 2*?w))
:- shape(?t, ?w, ?w), s(?i, ?v), REM(?i, 2 * ?w) < ?w .

compare(?t, ?i, ?i + ?s)
:- shape(?t, ?w, ?s), s(?i, ?v), REM(?i, 2 * ?s) < ?s, ?s < ?w .

% sorting(?t, ?p_original, ?p) zum Zeitschritt ?t ist der wert,
% welcher anfangs am Index ?p_original war, nun am Index ?p
sorting(0, ?p, ?p) :- s(?p, ?v) .

comparison(
    ?t, ?p1, ?p2, ?op1, ?op2,
    2*COMPARE(?v1, ?v2) - COMPARE(STR(?op2 - ?op1), ".")
)
:- compare(?t, ?p1, ?p2), sorting(?t, ?p1, ?op1),
   sorting(?t, ?p2, ?op2), s(?op1, ?v1), s(?op2, ?v2) .

sorting(?t+1, ?p1, ?op1 * ?r + ?op2 * (1-?r)),
sorting(?t+1, ?p2, ?op2 * ?r + ?op1 * (1-?r))
:- comparison(?t, ?p1, ?p2, ?op1, ?op2, ?rp), lut(?rp, ?r) .

sorting(?t+1, ?p, ?op)
:- sorting(?t, ?p, ?op), compare(?t, ?p, ?p2), len(?l), ?p2 >= ?l .

% extrahieren der Sortierung aus dem letzten Zeitschrit
end(#max(?t) + 1) :- shape(?t, ?w, ?s) .
sorted(?p, ?op, ?v) :- end(?t), sorting(?t, ?p, ?op), s(?op, ?v) .
```

Bitonic-merge-sort hat jedoch eine etwas schlechtere Laufzeit als `n*log(n)` außerdem wird etwas Zeit aufgewendet in den Prädikaten `shape` und `compare` um das Sorting-Network aufzubauen bevor der eigentliche Sortiervorgang beginnt. Dennoch ist es das schnellste bekannte vergleichsbasierte Sortierverfahren in Nemo. Es benötigte 8,6 Sekunden in der Browser Version von Nemo auf 10.000 Eingabewerten. Ein weiterer Nachteil ist, dass eine initiale Reihenfolge gegeben sein muss, zwar lässt sich eine solche in Nemo mit dem oben

vorgestellten Hacker-Mans-Sort für beliebige Eingabetypen generieren, jedoch kommt eine saubere Lösung, wie das als nächstes vorgestellte Radix-sort, ohne initiale Reihenfolge aus.

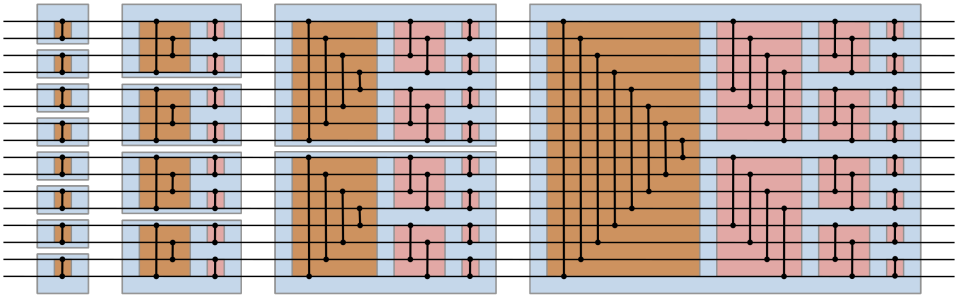

Abbildung 3.3: Visualisierung des Bitonic Sort Netzwerks
https://en.wikipedia.org/wiki/Bitonic_sorter#Alternative_representation

Radix Sort

Radix-sort stellt jeden zu sortierenden Wert in einer gemeinsamen Baumstruktur dar, wobei Werte mit gemeinsamen Präfix sich den entsprechenden Anfang des Pfades von der Wurzel zum Blatt, welches für den Wert steht, teilen. Die vorgestellte Implementierung wurde umfangreich optimiert. Es wird nicht für jeden Character ein neuer Knoten in der Baumstruktur erzeugt. Stattdessen wird, für den gerammten Wert, zunächst nur ein Knoten erzeugt und Werte werden, nur wenn erforderlich, rekursiv in Präfix und Suffix halbiert. Ein Split ist erforderlich, wenn ein Knoten mehrere Kindknoten hat. Dies wirkt zunächst schwierig ohne Aggregation, welche rekursiven Aufbau der Baumstruktur wegen nicht stratifiziertem Programm verhindern würde, zum Zählen der Kindknoten. Folgender Pattern in Nemo ist jedoch erstaunlich effizient in der Verwendeten Nemo Version (nicht quadratisch in Laufzeit): `multiple_values(yes) :- values(?a), values(?b), ?a != ?b ..`

Hier der Radix-Sort Algorithmus in Nemo. Es wurden wieder erklärende Kommentare eingefügt:

```
% Die maximale Länge definiert gemeinsame Split punkte
% der Strings; aufrunden auf Zweierpotenz wäre ineffizienter
l(#max(STRLEN(?s))) :- s(?s) .



% === BAUEN DES RADIX BAUMES ===
% Zu Beginn ist jeder Eingabestring ein Kind der Wurzel ("")
% und geht von 0 bis zur maximalen Länge ?l
child("", 0, ?l, ?s) :- s(?s), l(?l) .

% Gibt es zu einem ?parent betrachtet von ?start bis
% ?end mehrere children?
multiple_children(?parent, ?start, ?end)
:- child(?parent, ?start, ?end, ?a),
   child(?parent, ?start, ?end, ?b),
   ?a != ?b .

% Ein ?parent von ?start bis ?end ist in die knoten von
```

```
% ?start bis mid und von mid bis ?end zu teilen, wenn er
% mehrere Kinder hat
split(?parent, ?start, ?start + (?end - ?start) / 2, ?end)
:- multiple_children(?parent, ?start, ?end),
   ?end - ?start >= 2 .

% Ein zu splittender String wird gesplittet
child(?parent, ?start, ?mid, ?mid_str),
child(?mid_str, ?mid, ?end, ?child)
:- child(?parent, ?start, ?end, ?child),
   split(?parent, ?start, ?mid, ?end),
   ?mid_str = SUBSTRING(?child, 1, MIN(?mid, STRLEN(?child))) .

% Welche Strings wurden später nicht mehr gesplittet?
min_end(?parent, ?start, #min(?end))
:- child(?parent, ?start, ?end, ?child) .

true_child(?parent, ?child)
:- min_end(?parent, ?start, ?end),
   child(?parent, ?start, ?end, ?child),
   ?parent != ?child .



% === ZÄHLEN UND SORTIEREN DER KINDER ===
% jeder Eingabe-String wird zu seinem Knoten gezählt
at(?s, ?s) :- s(?s) .

% Jeder Eingabe-String wird zusätzlich zu seinem Parent gezählt
at(?parent, ?s) :- at(?child, ?s), true_child(?parent, ?child) .

% Zählen der Strings an einem Knoten
% dies ermöglicht später das Generieren des Indexes mit
% deutlich weniger Regelaufrufen (in "parallel")
count(?node, #count(?s)) :- at(?node, ?s) .

% sortieren der Kinder durch "sort by counting"
child_index(?parent, ?child, #count(?other))
:- true_child(?parent, ?child),
   true_child(?parent, ?other),
   COMPARE(?other, ?child) <= 0 .



% === GENERIEREN DES INDEX ===
% Der erste String unter der Wurzel hat den Index 0
index(0, "") .

% Das erste Child hat den Index wie das Parent
% wenn das Parent keiner der Eingabe Strings ist
index(?i, ?child)
```

```
:- index(?i, ?parent),
   child_index(?parent, ?child, 1),
   ~s(?parent) .

% Das erste Child hat den nächsten Index nach dem Parent
% wenn das Parent einer der Eingabe-Strings ist
index(?i + 1, ?child)
:- index(?i, ?parent),
   child_index(?parent, ?child, 1),
   s(?parent) .

% Das nächste Kind hat den Index, der um die Anzahl der
% Strings unter dem ersten Kind größer ist.
% Dadurch, dass die Anzahl der Strings unter dem ersten Kind
% bereits bekannt ist, kann die Indexberechnung unter dem
% nächsten Kind direkt beginnen, was den Vorgang erheblich
% effizienter macht
index(?i + ?c, ?next)
:- index(?i, ?before),
   count(?before, ?c),
   child_index(?parent, ?before, ?child_i),
   child_index(?parent, ?next, ?child_i+1) .

% filtern aller Indices nach solchen, die für das Ergebnis
% relevant sind
sorted(?i, ?s) :- index(?i, ?s), s(?s) .
```

Dieser Algorithmus benötigt nur 46 Sekunden für 1 Mio. Strings. Für 100.000 Strings benötigt er 4,4 s, was die annähernd lineare Laufzeit des Algorithmus zeigt. Es besteht eine signifikante Abhängigkeit der Laufzeit von der Länge der Strings. Haben alle Strings die Länge 100, werden 100.000 Strings in 10 Sekunden sortiert. Gleiche Präfixe bzw. Suffixe haben in dieser Form des Algorithmus keinen großen Einfluss auf die Laufzeit.

SPARQL compliant Sort

Der Radix-sort Algorithmus kann als Grundlage genutzt werden, einen Sortieralgorithmus, welcher alle Datentypen korrekt sortiert, zu entwickeln. Dieser ist jedoch in seiner Rohfassung über 100 Zeilen lang (zum vergleich der Radix-sort Algorithmus ist in seiner nicht für die Arbeit angepassten Formatierung 21 Zeilen lang) und wird deswegen hier nicht vollständig abgebildet.

Zentraler Punkt im Algorithmus ist eine „Sortiertabelle“ diese hat zwei Zeilen für jede zu sortierende Variable und eine Zeile für jedes zu sortierende Tupel. Beispielsweise bei `ORDER BY ?a ?b` hätte sie 5 Zeilen. Dabei wird für jede Variable die erste Spalte für einen Zahl basierend auf dem Datentyp und die zweite Spalte für eine Zahl zum Sortieren innerhalb des Datentyps verwendet. Die Sortierung der Eingabe basiert auf Sortierung der „Sortiertabelle“ nach der ersten Spalte, wobei bei Gleichheit einer Zeile bis zur *n*-ten Spalte die nächste Spalte zum Sortieren verwendet wird. Die letzte Spalte der Sortiertabelle ist der ursprüngliche Index, was das Sortierverfahren stable macht und dafür sorgt, dass keine Spaten am Ende denselben Index zugewiesen bekommen, statt dem Index kann auch ein

beliebiges anderes Unterscheidungsmerkmal, wie der Name eines neu generierten BNodes, verwendet werden.

Hier die Schritte des Algorithmus:

- Aufteilen der Eingabewerte in numerisch und nicht-numerisch, alle nichtnumerischen Werte werden zunächst als String konvertiert
- Sortieren aller nicht-numerischen Werte mittels Radix-sort
- Bilden der „Sortiertabelle“
- Konvertieren der Zahlen zu Strings und korrektes Einrücken (Alignment der dezimalen einer-Position)
- bilden eines Strings für jede Zeile der „Sortiertabelle“
- Sortieren der Strings aus der „Sortiertabelle“ mittels Radix-sort
- Transformieren der Indices im zu sortierenden Prädikat mittels der sortierten Strings aus der „Sortiertabelle“

Alle nicht numerischen Strings werden gemeinsam sortiert, der resultierende Index bildet die Zahl nach der Werte des gleichen Typs untereinander verglichen werden in der „Sortiertabelle“. Zum Einrücken der Zahlen wird zunächst die Maximallänge aller Zahlen als String in einer Spalte bestimmt. Die Länge einer Zahl wird dabei nur bis zum Komma gemessen, was mit dem Ausdruck `STRLEN(STRBEFORE(CONCAT(?s, ".“), ".“))` für eine als String konvertierte Zahl `?s` berechnet werden kann. Positive Zahlen werden mit „0“ und negative Zahlen werden mit „-“ bis zur maximalen Länge aufgefüllt, außerdem werden alle negativen Zahlen in ihrer Reihenfolge invertiert. „-“ kommt alphabetisch vor allen Zahlen, wodurch negative Zahlen vor positiven sortiert werden. ” ” (Space) kommt alphabetisch vor „.“ (Komma). Dies bewirkt, dass Zahlen mit Nachkommastellen nach Zahlen ohne Nachkommastellen sortiert werden, wenn die nächste Zahl der Spalte in der „Sortiertabelle“ durch ” ” (Space) getrennt ist. In der letzten Spallte würde dies gelten, da Zahlen mit Komma länger sind, jedoch ist die letzte Spalte der ursprüngliche Index, welcher immer eine nicht negative ganze Zahl ist. Generell können unterschiedliche Zeilen der Strings für die Zeilen der „Sortiertabelle“ unterschiedlich lang sein, wegen unterschiedlicher Nachkommalängen. Dies ist jedoch kein Problem, da nur der Vergleich von Zeilen mit gleichem Präfix und damit gleicher bisheriger Nachkommalänge relevant ist. Schließlich ist anzumerken, dass boolesche Werte in Nemo als nicht numerisch angesehen werden „true“ kommt jedoch korrekt nach „false“ alphabetisch.

Die Übersetzungsimplementierung, welche diese Nemo-Programme für beliebige `OrderBy` Operationen in SPARQL-Queries generiert, umfasst 180 Zeilen. Dabei entfallen 17 Zeilen auf die Berechnung des Type-Scores `type_score()` welcher dafür sorgt, dass der *UNDEF-Marker*↗ als erstes, dann BNodes, IRIs, Zahlen, dann Strings und zum Schluss alle verbleibenden Werte sortiert werden. 48 Zeilen entfallen auf die Implementierung des Radix-sort Algorithmus `radix_sort()`, 105 Zeilen auf den Sortieralgorithmus wie oben beschreiben `translate_order_by_seq()` und 10 Zeilen auf das Aufrufen der Implementierung im richtigen fall und Transformation der Ergebnisse für Ergebnistypen außer `SolutionSequence`.

3.8 Query-Formen

Es werden alle 4 SPARQL-Query-Formen `SELECT`, `DESCRIBE`, `ASK` und `CONSTRUCT` unterstützt. Spargebra unterstützt dabei, indem bereits die korrekte Projektion als SPARQL-Algebra-Opperation generiert wird. Bei `SELECT` sind dies die selektierten Variablen. Bei `DESCRIBE` ist dies die Variable, die die zu beschreibenden Werte enthält. In der Methode

`translate_query()` des `QueryTranslator` Objekts wird mithilfe eines `match` Statements die korrekte Funktion zum Übersetzen ausgewählt. Führ `SELECT` bei dem direkten Kindknoten in der SPARQL-Algebra ein `DISTINCT` ist, oder ein `SLICE` von einem `DISTINCT`, wird die `translate_pattern()` Methode direkt verwendet. Bei allen anderen `SELECT` Queries wird die `translate_pattern_seq()` Methode verwendet. Die anderen Query Formen greifen auf die `translate_pattern()` Funktion zurück. Diese Funktionen wurden bereits in Abschnitt 3.7 vorgestellt. Die Funktionen für `ASK` und `DESCRIBE` führen einfache Post-Prozession durch. Die `CONSTRUCT` Funktion ist deutlich komplexer.

3.8.1 Describe

Für die `DESCRIBE` Query ist im SPARQL Standard frei gestellt, wie die Ergebnisse der Query beschreiben werden sollen. Die Implementierung gibt einfach alle Triples, die einen der zu beschreibenden Knoten in Subject position haben:

```
for v in get_vars(&solution){
    nemo_add!(output_graph(v, ?p, ?o) :- input_graph(v, ?p, ?o), {&solution});
}
```

`solution` wurde zuvor durch die `translate_pattern()` Methode erzeugt, `output_graph` ist ein zuvor erzeugtes dreistelliges Prädikat und `get_vars()` liefert alle Attribute des gegebenen Prädikats.

3.8.2 Ask

`ASK` gibt einen Boolean zurück, der angibt, ob die zugrundeliegende Query ein Ergebnis hat. Dies kann mit stratifizierter Negation in Nemo umgesetzt werden:

```
nemo_add!(dummy(0));
nemo_def!(ask(true) :- {&solution}; SolutionSet);
nemo_add!(ask(false) :- dummy(0), ~{&solution});
```

Das `dummy` Prädikat muss erstellt werden, da Nemo in der Aktuellen Version keinen *Regel-Körper*↗ unterstützt, welcher nur aus negierten Atomen besteht. Das `solution` Prädikat wurde zuvor durch die `translate_pattern` Methode erstellt.

3.8.3 Construct

Die `CONSTRUCT` Query ist in der Übersetzung die Komplexeste Query Form, da sie BNodes im Resultierenden Graphen unterstützt. Diese müssen zunächst erzeugt werden. Außerdem schreibt der SPARQL Standard vor, dass im Triples, die nach RDF nicht valide sind, aus dem resultierenden Graph gefiltert werden müssen, insbesondere sind in subject Position nur BNodes und IRIs und in Prädikat Position nur IRIs erlaubt. Die eigentliche Konstruktion des Graphen geschieht in einer Regel für jedes Triple im gegebenen Pattern. Hier ein Beispiel:

Query:

```
prefix ex: <https://example.org/>
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

CONSTRUCT {
    [a ex:Relation; rdf:subject ?s; rdf:object ?o]
} WHERE {
    ?s ex:rel ?o .
}
```

Nemo-Programm (manuell BNODE_VAR unbenannt, Präfixe und Kommentare eingefügt):

```
@prefix ex: <http://example.org/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

% the query
bgp(?s, ?o) :- input_graph(?s, ex:rel, ?o) .

% generate bnodes for each result
bnode_solution(?s, ?o, !BNODE_VAR) :- bgp(?s, ?o) .

% generate triples for each pattern triple
proto_graph(?BNODE_VAR, rdf:type, ex:Relation)
:- bnode_solution(?s, ?o, ?BNODE_VAR) .

proto_graph(?BNODE_VAR, rdf:subject, ?s)
:- bnode_solution(?s, ?o, ?BNODE_VAR) .

proto_graph(?BNODE_VAR, rdf:object, ?o)
:- bnode_solution(?s, ?o, ?BNODE_VAR) .

% filter invalid triples
construct(?s, ?p, ?o) :-
    proto_graph(?s, ?p, ?o),
    AND(OR(isNull(?s), isIri(?s)), isIri(?p)) = "true"^^xsd:boolean .

@output construct .
```

Zum Extrahieren der BNodes aus dem Pattern und zum Generieren der Regeln mit den entsprechenden Konstanten aus dem Pattern wird die `translate_pattern()` Methode verwendet. Diese ist bereits in Unterabschnitt 3.7.2 vorgestellt.

4 Evaluation

In diesem Kapitel wird die in Kapitel 3 vorgestellte Implementierung hinsichtlich ihrer Korrektheit in Bezug auf den SPARQL Standard (Abschnitt 4.1) und ihren Skalierbarkeit (Abschnitt 4.2) evaluiert.

4.1 Evaluierung der SPARQL Konformität

Um die SPARQL Konformität zu testen, wurden die Queries der SPARQL 1.1 Test Suite [8] ausgeführt und mit den erwarteten Ergebnissen verglichen. Die SPARQL 1.1 Test Suite wurde von der W3C SPARQL Working Group zusammengetragen. Sie umfasst unterschiedliche Gruppen von Tests, wie z.B. eine Gruppe für Tests des Subquery-Features in SPARQL. Zu jeder Gruppe sind Metadaten in Form einer `manifest.ttl` Datei gegeben. Diese enthält auch Informationen über jeden einzelnen Test-Case, welche dazu verwendet werden können, eine Auswahl von relevanten Tests zu treffen. Beispielsweise sind Update-Queries und Tests unterschiedlicher Ergebnis-Serialisierungen nicht Bestandteil dieser Arbeit.

Folgende RDF Präfixe der Test Suite sind für die Auswahl der Test-Cases relevant:

Prefix	Namespace-IRI
`rdf:`	http://www.w3.org/1999/02/22-rdf-syntax-ns#
`rdfs:`	http://www.w3.org/2000/01/rdf-schema#
`mf:`	http://www.w3.org/2001/sw/DataAccess/tests/test-manifest#
`qt:`	http://www.w3.org/2001/sw/DataAccess/tests/test-query#
`sd:`	http://www.w3.org/ns/sparql-service-description#

Es werden alle Test-Cases vom Typ `mf:QueryEvaluationTest` ausgeführt, welche alle der folgenden Bedingungen erfüllen. Als „Test-Action“ wird dabei der Wert der Property `mf:action` bezeichnet:

- die Test-Action hat keine weiteren Eingabegraphen außer den Standard Graph (`qt:graphData`)
- die Test-Action referenziert kein Entailment-Regime (`sd:entailmentRegime`)
- drei Testgruppen wurden manuell von den Tests ausgeschlossen: „Service“, „JSON Result Format“, „CSV/TSV Result Format“

Außerdem ergeben sich einige weitere Bedingungen durch Features, die nicht von der Testimplementierung unterstützt werden:

- die Test-Action referenziert Eingabewerte (`qt:data`)
- die erwarteten Ergebnisse (`fm:result`) sind nicht vom Typ „.ttl“ - keine Unterstützung von Construct Queries

Ein Test-Case wurde ausgeschlossen, da die erwarteten Ergebnisse in der Testsuite fehlten.

Die Durchführung der Tests wurde in Python implementiert. Die Implementierung wird hier kurz zusammengefasst und jeweils die Zeilenanzahl für den jeweiligen Programmteil genannt, um einen groben Einblick der Komplexität zu geben. Generell ist die Implementierung einfach gehalten und umfasst 344 Zeilen. Davon entfallen 77 Zeilen auf das Lesen und Interpretieren der Test Suite und die Auswahl der Tests. In 8 Zeilen ist eine Transformation der Eingabedaten implementiert, welche dazu verwendet werden, Datentypen mit limitierter Unterstützung in Nemo umzuwandeln. Das eigentliche Ausführen der Programme wurde in 108 Zeilen implementiert. Das zu testende Programm, welches die Übersetzung von SPARQL zu Nemo durchführt, wurde dazu in einem Modus kompiliert, welcher es erlaubt, die SPARQL-Query mittels stdin zu übergeben. Damit das Ergebnis der Übersetzung zu einem vollständigen Nemo-Programm wird, ist das Laden der Eingabedaten und das Exportieren des korrekten Ergebnisprädikats, dem Nemo Code hinzuzufügen. Als Rule-Engine wird Nemo durch das Nemo Command Line Interface aufgerufen. In 82 Zeilen werden die Ergebnisse aus Nemo mit den erwarteten Ergebnissen verglichen. Um den Vergleich durchzuführen, werden die durch Nemo in CSV-Format exportierten Ergebnisse zur RDF Repräsentation der Python Bibliothek rdflib [5] transformiert und die erwarteten Ergebnisse ebenfalls mittels rdflib geparst, wodurch ein Vergleich der gleichen Repräsentationen möglich wird. Falls die Query den String „`ORDER BY`“ enthielt, wird die Reihenfolge der Zeilen überprüft, ansonsten findet ein Vergleich ohne Beachtung der Reihenfolge statt. Enthält die Query den String „`select *`“, ohne Beachtung der groß und Kleinschreibung, wird die Ordnung der Spalten nicht überprüft. Für das Sammeln und Ausgeben des Test-Reports werden 38 Zeilen Code verwendet. Anschließend werden die fehlerhaften Queries manuell analysiert.

Um sicherzustellen, dass das zu testende Programm nicht speziell für die Test-Suite optimiert ist, und sich die Ergebnisse somit nicht verallgemeinern lassen, wird idealerweise das zu testende Programm nach der ersten Ausführung der Evaluation nicht mehr verändert. Allerdings geht dadurch das Potenzial verloren, durch die Tests identifizierte Fehler zu beheben, was zu einem insgesamt besseren Programm führen könnte. Als Kompromiss wird hier der Prozess der Evaluation mit initialen Ergebnissen, daraus resultierenden Änderungen am Programm und wiederum daraus resultierenden Ergebnissen der Evaluation vorgestellt.

Zunächst die initialen Ergebnisse, wie diese bei der Testausführung generiert wurden:

```
Total: 169
Failed: 35
Errored: 86
    MultiPatternNotImplemented(Group)  - 26
    SequencePatternNotImplemented(Minus)  - 5
    MultiPatternNotImplemented(Minus)  - 1
    FunctionNotImplemented - 26
    AggregationNotImplemented  - 3
    MultiPatternNotImplemented(Path)  - 15
    ParseError  - 10
Success: 48
Empty Groups: 22
```

Es erfolgt die Erklärung der einzelnen Anzahlen im Test-Report:

- `Total`: Anzahl der von der Testimplementierung als relevant ausgewählten Test-Cases
- `Failed`: Test-Cases, bei denen das erwartete und das tatsächliche Ergebnis nicht übereinstimmen
- `Errored`: Test-Cases, bei denen die Übersetzung der Query einen Fehler verursachte. Die genauen Fehler werden dabei in Typen kategorisiert. Ein Fehler-Typ ergibt sich aus der Fehlermeldung bis zum ersten „:“ Zeichen. Diese Fehler-Typen werden mit der jeweiligen Häufigkeit einzeln aufgelistet. „`FunctionNotImplemented`“ und „`AggregationNotImplemented`“ Fehler wurden zu einem Fehler-Typ zusammengefasst
- `Success`: Erfolgreiche Test-Cases
- `Empty Groups`: 22 Gruppen testen ausschließlich Features, welche von der Test-Implementierung als nicht relevant verworfen wurden, daher enthalten diese Gruppen keine Tests

Die zehn Fehler beim Parsing der SPARQL-Query (`ParseError`) werden direkt von der verwendeten spargebra Bibliothek beim Transformieren in SPARQL-Algebra verursacht. Die entsprechenden Queries sind tatsächlich nicht syntaktisch korrekt und nicht offiziell Teil der Test-Suite. Dies wurde jedoch nicht bei der Testauswahl erkannt. Alle weiteren Errors sind auf nicht implementierte Features zurückzuführen. Die nicht implementierten Funktionen sowie die nicht implementieren Aggregationsfunktionen `GroupConcat` und `Sample` sind dabei absichtlich, da diese nicht von Nemo unterstützt werden. Die Implementierungen mit Beachtung der Multiplizität bzw. Reihenfolge (ohne `DISTINCT`, genannt „MultiPattern“ bzw. „SequencePattern“) für Group, Minus und Path wurden jedoch bei der Implementierung übersehen und wurden deshalb direkt nachgeholt, und wurden bereits in Kapitel 3 erläutert. Während der Implementierung dieser Features wurden nicht die Queries der Evaluation betrachtet, sondern wie bei allen anderen Features die Rust-Tests, wie in Abschnitt 2.5 beschrieben, erweitert. Außerdem wurde erst nach diesem initialen Test Durchlauf die Nichtbeachtung der Spaltenreihenfolge für Queries mit `SELECT *` beim Vergleichen der Ergebnisse implementiert. Durch diese Änderungen ergibt sich folgendes Ergebnis. Die Statistiken für leere Gruppen ändern sich nicht und werden daher nicht angegeben:

```
Total: 169
Failed: 41
Errored: 46
    FunctionNotImplemented - 26
    AggregationNotImplemented(GroupConcat)  - 3
    AggregationNotImplemented(Avg)  - 4
    AggregationNotImplemented(Sample)  - 3
    ParseError - 10
Success: 82
```

Es ist zu beobachten, dass damit alle Fehler durch fehlende Features in Nemo erklärt werden können. Die `Failed` queries wurden manuell analysiert und nach Sorten von Fehlern eingeteilt:

- 16 Fehler: fehlende Unterstützung von Strings mit Language Tags in Nemo-Funktion
- 13 Fehler: fehlerhafter Umgang mit Datentypen in Nemo
- 7 Fehler: Bug in der ORDER-BY Übersetzung
- 2 Fehler: 0-Tupel statt leere Ergebnisse
- 2 Fehler: Fehler in der Testimplementierung

Der Bug in der ORDER-BY Übersetzung wurde darauf hin behoben. Die meisten Fehler resultieren durch fehlende Implementierung für bestimmte Literale in Nemo. Da dies nicht die Übersetzung von SPARQL nach Nemo, sondern Nemo selbst evaluiert, wurde eine „Soft" Modus der Test-Cases eingeführt. Ein Test Case ist im Soft-Modus erfolgreich, wenn er erfolgreich ist, nachdem im Eingabe-Graph alle Zahlen zu `xsd:double` konvertiert werden und alle Strings zu Plain-Strings konvertiert werden. Da die erwarteten Ausgabedaten damit nicht mehr gültig sind, wurden neue erwartete Ausgabedaten für den Soft-Modus durch Ausführen der Query mittels rdflib generiert. Dieser Soft-Modus sorgte für eine bessere Evaluation der Übersetzung, da einige Fehler durch Fehler in Nemo verdeckt wurden. In der Übersetzung konnte so ein Fehler im Umgang mit dem *UNDEF-Marker*↗ in der `MINUS` Implementierung gefunden und behoben werden. Im folgenden ist der finale Test-Report im Soft-Modus dargestellt:

```
Total: 169
Failed: 14
Errored: 46
    FunctionNotImplemented - 26
    AggregationNotImplemented - 10
    ParseError  - 10
Success: 109
```

Die 14 Fehler wurden wieder manuell analysiert und kategorisiert:

- 1 Fehler: Fehlende Unterstützung von Strings mit Language-Tags in Nemofunktion
- 5 Fehler: Fehlerhafter Umgang mit Datentypen in Nemo
- 3 Fehler: 0-Tupel statt leere Ergebnisse
- 3 Fehler: Fehler in der Testimplementierung
- 2 Fehler: Bug in der `STRBEFORE` Implementierung in Nemo

Es ist zu beobachten, dass die Fehler durch den Umgang mit Literalen in Nemo deutlich weniger geworden sind, diese jedoch dennoch auftreten, da nicht jedes Literal aus den Eingabedaten kommt und somit der Soft-Modus keinen Effekt hat. Die drei Fehler in der Testimplementierung kommen daher, da BNodes beim Vergleich der tatsächlichen Ergebnisse mit den vorgegebenen Ergebnissen nicht richtig zugeordnet wurden, die Ergebnisse in diesen drei Fällen sind jedoch korrekt. Die zwei Fehler in `STRBEFORE` treten durch fehlerhafte Unicode Unterstützung in dieser Funktion auf. `STRBEFORE("français", "s")` ergibt den String `"França"` mit fehlendem „i" in der verwendeten Version von Nemo. In drei Fehlerfällen ist das scheinbar erwartete *Ergebnis*↗ eine leere Menge jedoch das tatsächliche Ergebnis eine einelementige Menge mit dem leeren Tupel. Dies konnte auf einen Fehler in rdflib zurückgeführt werden, welche das Query Ergebnis in diesem Sonderfall nicht korrekt einliest, die Übersetzung ist korrekt. Insgesamt 6 der 14 Fehler sind also keine Fehler in der Übersetzung. Alle verbleibenden 8 Fehler sind Fehler in Nemo bzw. könnten durch einen verbesserten Soft-Modus gelöst werden.

Die ermittelten Ergebnisse können unterschiedlich interpretiert werden. Zum einen wurde, mit dem beschriebenen Verlauf, die Implementierung für die Testsuite optimiert. Dadurch ist theoretisch nicht auszuschließen, dass auf einer anderen Test Suite ähnlich viele Fehler wie bei den initialen Testergebnissen auftreten würden. Die SPARQL 1.1 Test Suite deckt andererseits einen signifikanten Anteil der SPARQL Features ab, auch wenn einige in Kapitel 3 erwähnte Abweichungen vom Standard nicht gefunden wurden. Sodass, als Ergebnis der Evaluierung, die Übersetzung der SPARQL-Queres im weitgehend als standardkonform angesehen werden kann. Die Laufzeit aller 169 Test-Cases beträgt etwa 7 Sekunden.

4.2 Evaluierung der Skalierbarkeit

Zum Evaluieren der Skalierbarkeit wurden Wikidata Beispielqueries übersetzt und mit Nemo ausgeführt. Da Nemo alle Daten vollständig im Arbeitsspeicher vorhält, ist dafür, aufgrund der Größe von Wikidata, ein spezielles Testsystem notwendig. Das verwendete Testsystem verfügt über 768 GiB RAM (Quad-Channel DDR4, 1866 MT/s) und zwei Intel Xeon E5-2637 v4 CPUs. Weitere Spezifikationen siehe Anhang. Als Eingabe Graph wurde der `latest-truthy` Wikidata Dump vom 22.11.2024 genutzt und daraus alle Triples entfernt, welche nicht-englische Language Tags beinhalten. Dies führt zu einem Datensatz mit etwa 3,3 Milliarden Tripel.

Das Programm zum Ausführen der Tests ist grundlegend ähnlich zu dem in Abschnitt 4.1 vorgestellten. Jedoch wurden einige zusätzliche Features erforderlich:

- Das Programm zum Ausführen der Testsuite ist in der Lage nach einem Crash oder absichtlichen Beenden die Arbeit wieder aufzunehmen. Dabei wird mit der nächsten Query nach der Query, welche den Crash verursachte begonnen.
- Es wurden zahlreiche Daten zur späteren Auswertung erfasst und inkrementell in json-Format in eine Datei geschrieben.
- Zum Vergleich der Ergebnisse werden die Queries außer in Nemo auch auf einer Blazegraph-Instanz (aufgesetzt, mit den im Quick_Start Guide[1] angegebenen Parametern) ausgeführt.
- Ein Python Script überwacht den Speicher und startet das Script, welches die Tests ausführt neu, falls zu viel Speicher verwendet wird.
- Ein separates Bash-Script schreibt alle 5 Minuten die Speichernutzung in eine Log-Datei zur späteren Nachvollziehbarkeit und dem Erkennen von Fehlern beim Neustarten des Programms.
- Ein Cron-Job sendet Benachrichtigungen über neu begonnene Queries. Zu lange laufende Queries wurden manuell (frühestens nach 30 Stunden) abgebrochen.
- Um eine gleichmäßige Verteilung der ausgeführen Queries zu gewehrleisten, wird jede 10. Query ausgeführt. Beim Starten des Programmes kann ein Offset angegeben werden, welche Query als erstes ausgeführt werden soll.

Die Wikidata Example Queries wurden über die Wikidata Wiki API angefragt und mittels des beautiful soup Paketes[2] extrahiert. Zu jeder Query wurde aus dem darüberliegenden Titeln ein Name generiert. Aus den ersten drei Wörtern des Titels und einem Hash wurde für jede Query eine ID, der „unique_name“ generiert. Die somit erstellte Test Suite umfasst 374 Queries. Damit die extrahierten SPARQL-Queries übersetzt werden können, müssen ihnen noch die Wikidata IRI-Präfixe angefügt werden. Außerdem müssen die `SERVICE` Operationen entfernt werden. Dies geschieht mit einem regulären Ausdruck. SPARQL ist keine reguläre Sprache, wodurch in komplexeren Fällen die `SERVICE` Operation nicht korrekt entfernt wurde.

[1] https://github.com/blazegraph/database/wiki/Quick_Start
[2] https://www.crummy.com/software/BeautifulSoup/bs4/doc/

Innerhalb etwa eines Monats wurden etwa ein Drittel, konkret 121 von 374, der Queries evaluiert. Davon wurden 22 Queries vorzeitig abgebrochen und lieferten somit keine Ergebnisse. 9 davon wurden aufgrund ihrer langen Ausführungszeit abgebrochen und die restlichen 13 aufgrund von hohem Speicherbedarf. Die restlichen 99 Queries lieferten Ergebnisse, wobei 62 davon erfolgreich waren und 37 einen Fehler beim Übersetzen lieferten. Im Folgenden sind die auftretenden Fehler beim Übersetzen dargestellt und in Kategorien eingeteilt:

- 10 Queries: Invalide Query durch Entfernen der `SERVICE` Operation
- 9 Queries: Fehlende Implementierung von Aggregationen (7x `Sample`, 2x `GroupConcat`)
- 8 Queries: Parse Error in spargebra (auch teilweise durch falsches entfernen der `SERVICE` Operation)
- 6 Queries: Fehlende Standard Funktionen (2x `Replace()`, 2x `Now()`, 1x `LangMatches()`, 1x `Rand()`)
- 4 Queries: Fehlende Funktionen, welche Erweiterungen des SPARQL Standards sind

Für die 62 erfolgreich ausgeführten Queries wurde die Korrektheit der Ergebnisse abgeschätzt, indem die Anzahl der Ergebnisse von Nemo und Blazegraph verglichen wurde. Bei 55 der 62 Queries gibt es übereinstimmende Ergebnisse, bei sechs Queries gibt es kein Blazegraph-Ergebnis und bei einer Query ist das Ergebnis falsch, da diese einen Vergleich zwischen Datum-Literals ausführt, welcher in Nemo nicht unterstützt wird. Somit lässt sich, auch unter Berücksichtigung der Evaluation in Abschnitt 4.1 abschätzen, dass etwa die Hälfte der Queries, welche getestet wurden, erfolgreich mit korrektem Ergebnis beantwortet werden konnten. Dies ist ein leicht besseres Ergebnis als auf den konstruierten Tests in Abschnitt 4.1 ohne Soft-Modus.

Etwa 10 Tage wurden für die erfolgreichen Queries zusammen an Nemo-Reasoning-Zeit benötigt. Davon wurden 8 Tage für das Laden der Eingabedaten benötigt. Die meiste Zeit wurde jedoch für die Queries verwendet, welche später abgebrochen wurden. Die Zeit, die das Übersetzen der Query in Nemo in Anspruch nimmt, ist vernachlässigbar. Sie betrug insgesamt, inklusive der Queries, die zu einem Fehler beim Übersetzen führten oder Abgebrochen wurden, 0,3 Sekunden.

Im Folgenden werden Ausführungszeit ohne Laden der Eingabedaten betrachtet. Das geometrische Mittel für die Ausführungszeit in Nemo beträgt 5,8 min (345 s) und für Blazegraph 273 ms. Aus dieser Perspektive ist die Query Ausführung in Nemo etwa 3 Größenordnungen langsamer. Die längste erfolgreiche Query in Nemo dauerte 11,6 stunden. Die längste erfolgreiche Query in Blazegraph dauerte 29,2 min. Es war für Blazegraph nicht nötig, eine Query wegen langer Dauer abzubrechen. Eine der Queries hängt nicht von den Eingabedaten ab und benötigt somit signifikant weniger Zeit. In Nemo benötigte diese 9 ms und in Blazegraph, inklusive Netzwerklatenz zu localhost, 12 ms.

Hier die Histogramme für die Ausführungszeiten der einzelnen Queries:

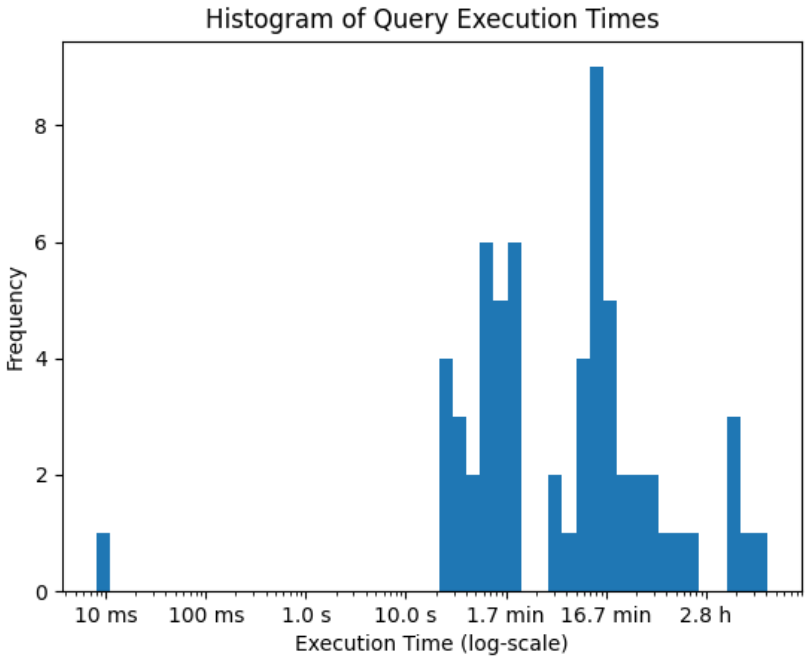

Abbildung 4.1: Histogramm der Ausführungszeiten der mit Nemo ohne Fehler ausgeführten Queries

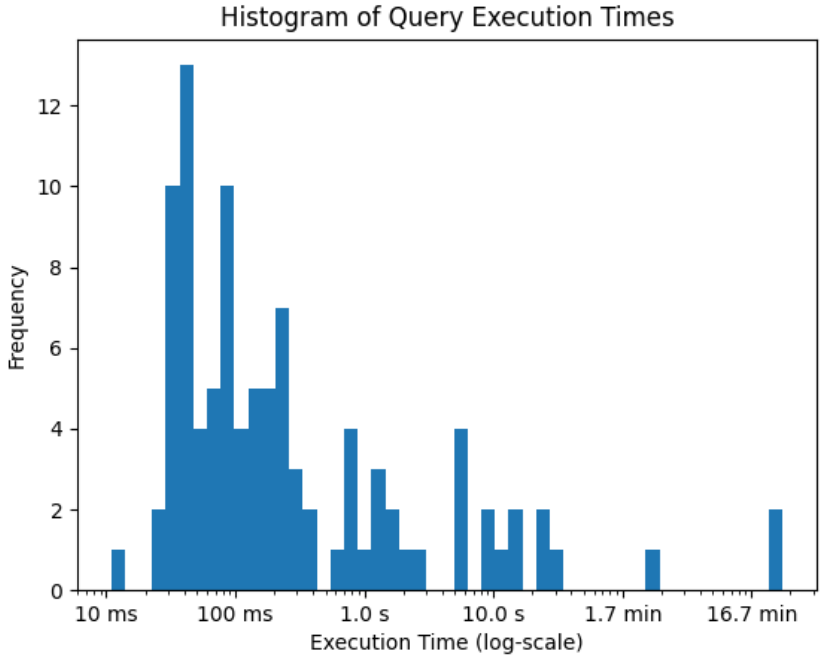

Abbildung 4.2: Histogramm der Ausführungszeiten der mit Blazegraph ohne Fehler ausgeführten Queries

Die Nemo-Zeitstatistiken wurden mit den Regeln zusammengeführt, aus deren Namen auf die SPARQL Features geschlossen werden konnte. Hier die Gesamtausfürungszeit der jeweiligen Features während aller erfolgreichen Queries:

Name	Zeit Gesamt	Anzahl mit Zeit $\geq$ 1 ms	Gesamtanzahl
`sort_string`	10 ms	2	16
`child_index`	12 ms	4	32
`left_join`	19 ms	2	30
`effective_boolean_value`	20 ms	7	56
`left_join_proto`	24 ms	3	30
`dummy_dependency`	30 ms	1	32
`has_child`	32 ms	6	64
`sum_aggregate`	36 ms	1	1
`count_aggregate`	39 ms	2	5
`collect_aggregations`	41 ms	2	7
`str`	46 ms	1	5
`regex`	49 ms	1	1
`index`	61 ms	12	96
`filter`	119 ms	4	21
`and`	371 ms	12	36
`lang`	473 ms	6	16
`equal`	741 ms	22	75
`if_expression`	742 ms	8	8
`extend`	958 ms	9	34
`multi`	2.3 s	3	14
`proto_extend`	2.5 s	10	34
`path_alternative`	5.9 s	11	12
`map`	12.4 s	10	41
`sequence_start`	32.7 s	4	47
`base_join`	42.1 s	4	24
`lcase`	53.3 s	1	4
`var`	1.1 min	13	73
`partial_exists`	1.1 min	7	30
`diff`	1.1 min	4	26
`recursive`	1.3 min	3	4
`reverse_recursive`	1.7 min	20	28
`exists`	1.8 min	9	60
`pre_index`	2.1 min	10	92
`sequence_shifted`	2.1 min	7	47
`sequence_proto`	3.2 min	7	47
`projection`	3.3 min	7	64
`proto_projection`	3.4 min	6	42
`union`	3.5 min	12	80
`sequence`	6.3 min	12	93
`one_or_more_path`	1.3 h	9	18
`join`	7.2 h	31	76
`path_property`	9.2 h	29	29
`bgp`	9.4 days	126	129

In der Tabelle ist der Name des Features (normalisierter Name des Prädikats im *Regel-Kopf* ↗, siehe Kapitel 3 für Erklärungen der Features), die Gesamtzeit des Features, die Anzahl, wie

oft das Feature mit einer Zeit über gerundet 0 ms und wie oft es insgesamt aufgetreten ist. Etwa die Hälfte der Features hat eine Gesamtzeit von unter 10ms und wird daher nicht in der Tabelle gelistet. Es ist anzumerken, dass Nemo die Importzeit der Daten mit zum jeweiligen Prädikat zählt, bei dem der Import stattgefunden hat. Dennoch ist nahezu die gesamte Ausführungszeit auf die Basic Graph Pattern Evaluierung zurückzuführen.

Die Beobachtung, dass die meiste Zeit durch Join-Opperationen des Input-Graphen mit sich selbst aufgewandt wird, motiviert folgende Grafik. Sie stellt die durchschnittliche Ausführungszeit einer Query in Abhängigkeit der Anzahl des Strings „input_graph“ im Programm dar:

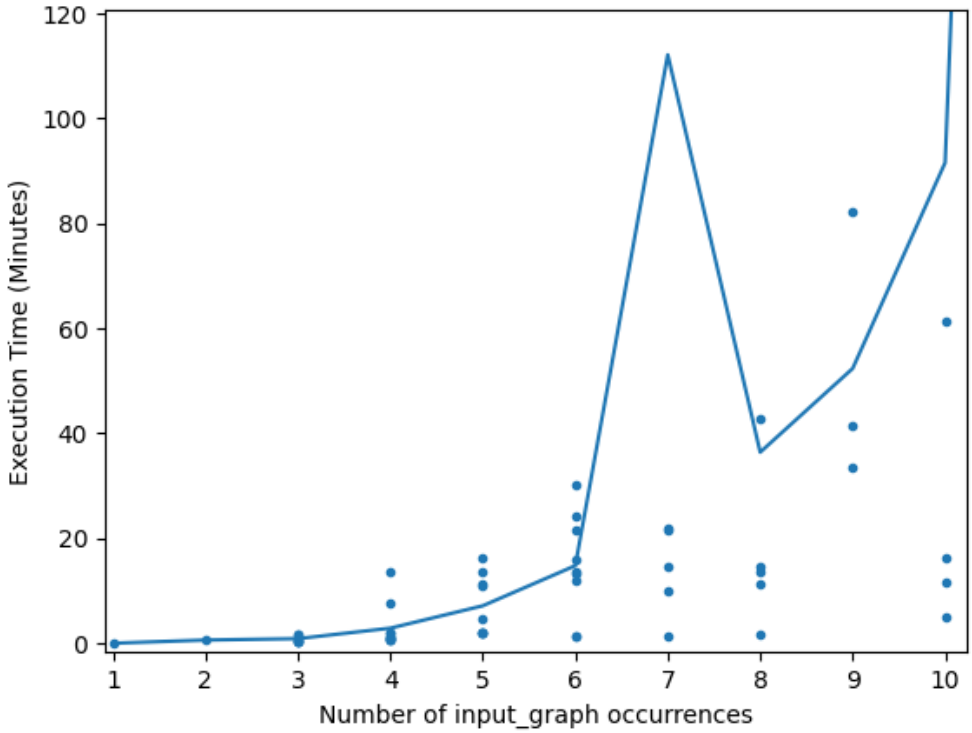

Abbildung 4.3: Durchschnittliche Ausführungszeit in Abhängigkeit der Anzahl, mit welcher der Eingabe Graph im Nemo-Programm verwendet wird

Die einzelnen Punkte stehen für die einzelnen Queries. Es wurden erfolgreiche Queries betrachtet. Es ist zu beobachten, dass sich im Durchschnitt eine gleichmäßige Kurve ergibt. Insbesondere für den Bereich bis etwa vier Verwendungen des input_graph Prädikats ist dies eine sehr gute Heuristik, um eine effiziente Query zu erkennen. Die Grafik ist in beide positiven Richtungen abgeschnitten. Das höchste Vorkommen an input_graph Prädikaten beträgt 11. Dort existiert ein weiterer Durchschnittswert, welcher deutlich über der sich ergebenden Kurve liegt, ähnlich wie bei 7. Bei 7 liegen zwei Queries über 120 min. Als besonderes Feature konnte für diese identifiziert werden, dass BNodes im Basic Graph Pattern enthalten und somit eine Aggregation zur Berechnung der Multiplizität in Nemo ausgeführt werden muss. Ob dies der tatsächliche Grund für die Spitze bei 7 ist, ist nicht endgültig geklärt. Die beiden lange laufenden Queries mit 7 Verwendungen des input_graph Prädikats werden wie die meisten anderen Queries auch von der Ausführung eines Basic Graph Patterns dominiert. Zufall wäre bei der geringen Datenlage und hohem Spread auch eine valide Erklärung.

Im Gegensatz zur Anzahl der input_graph Vorkommen ist die Anzahl der Ergebnisse der Query eher ein schlechter Indikator für die Laufzeit:

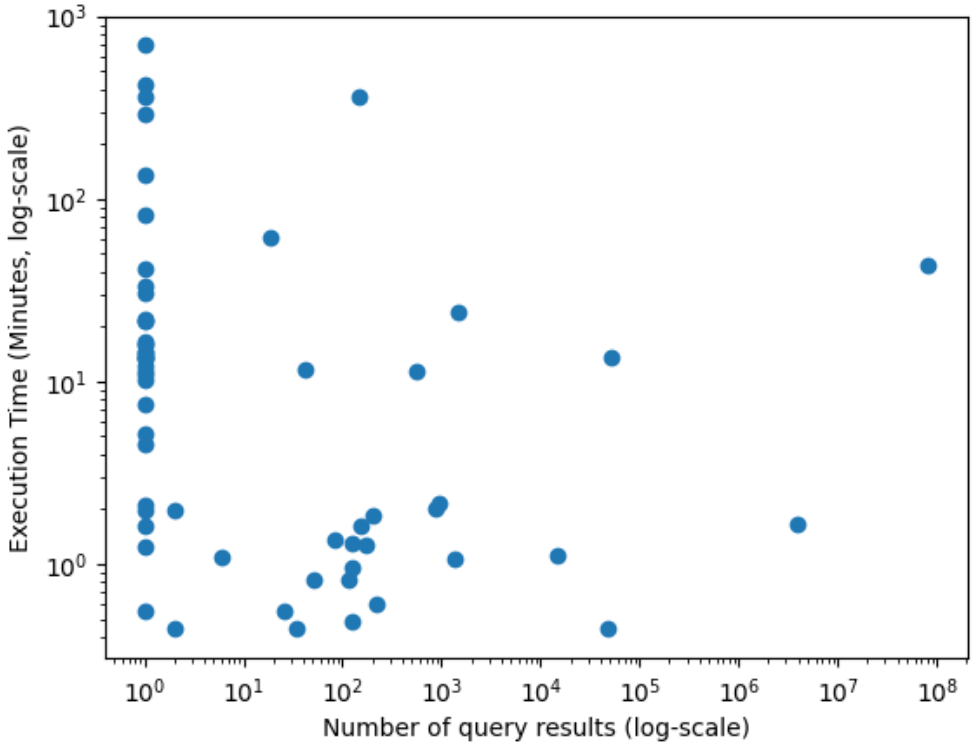

Abbildung 4.4: Keine Abhängigkeit zwischen Ausführungszeit und Anzahl der Ergebnissen

Um die Grafik zu erzeugen wurde die Query, welche nicht vom Eingabe-Graph abhängt, entfernt, da sie selbst auf der logarithmischen Achse stark von den restlichen Ergebnissen abweicht. Außerdem wurden alle Ergebnis-Anzahlen um eins erhöht, um die Queries ohne Ergebnisse auf der logarithmischen Achse darstellen zu können. Es ist zu beobachten, dass viele Queries keine Ergebnisse haben. Dies liegt daran, dass auf dem truthy Wikidata Graphen evaluiert wurde, welcher ein Subset des Gesamtgraphen darstellt und einige Prädikate nicht enthält. Dennoch ist zu beobachten, dass diese Queries teilweise lange Laufzeiten haben. Dies ist der Fall, obwohl manche im Basic Graph Pattern verwendeten Konstanten, insbesondere Prädikate, welche nicht im truthy Wikidata Graphen vorkommen, fast sofort zu einem leeren Ergebnis führen könnten.

Die bisherigen Betrachtungen haben sich auf die Ausführungszeit ohne Laden der Daten konzentriert. Hier das Histogramm für die Zeiten, für das Laden der Daten wie von Nemo angegeben:

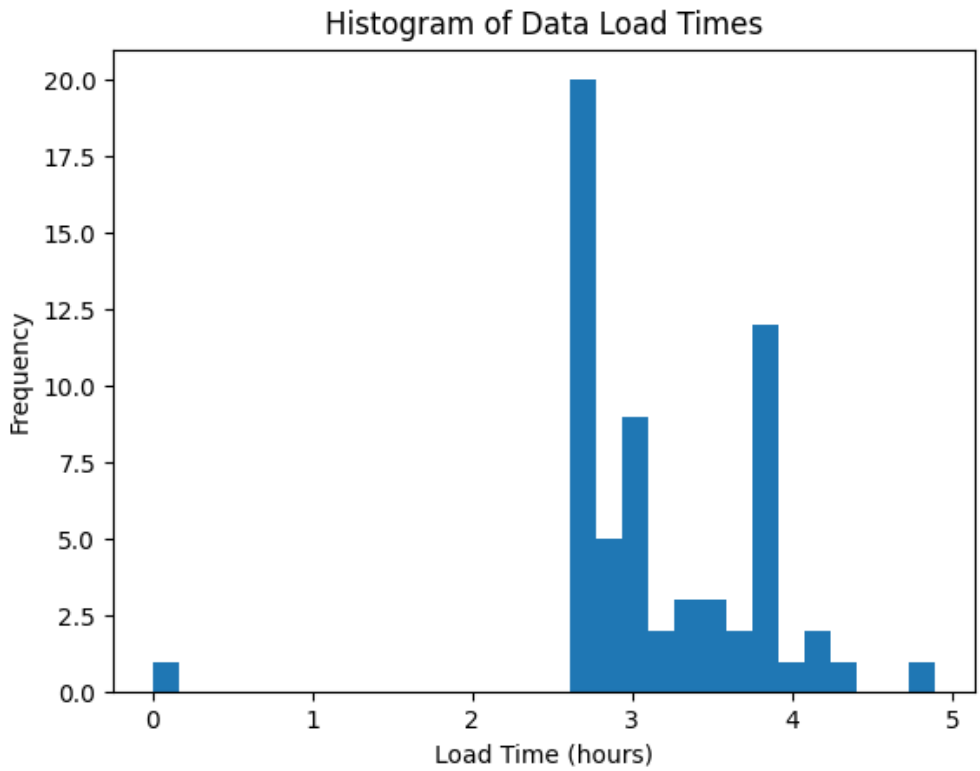

Abbildung 4.5: Histogramm der Ladezeiten des Eingabegraphen

Es ist zu sehen, dass das Laden der Daten etwa drei Stunden dauert. Zudem fällt wieder die Query, welche nicht von den Eingabedaten abhängt, auf. Dem gegenüber steht eine Ladezeit von drei Tagen in Blazegraph nach Befolgen des Quick Start Guides von Blazegraph auf dem oben beschriebenen Testsystem.

Zum Abschluss hier noch ein Ausschnitt aus der Visualisierung des belegten Speichers auf dem Testsystem in Abhängigkeit von der Zeit:

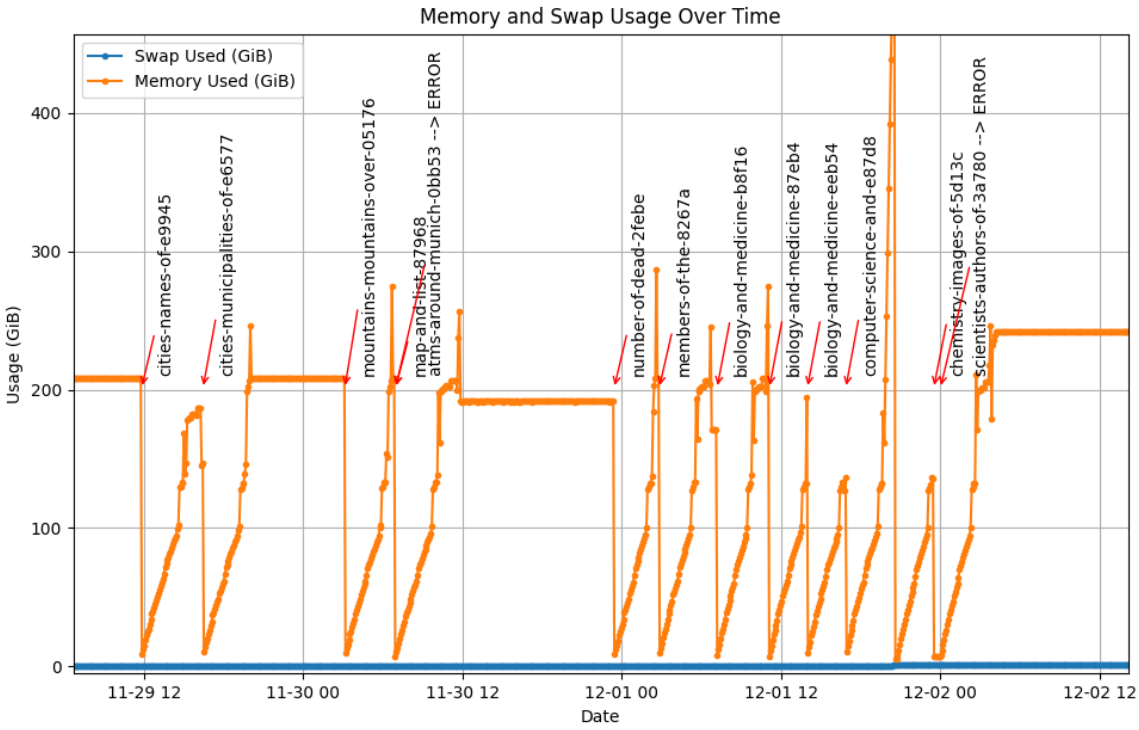

Abbildung 4.6: Ausschnitt aus dem Verlauf der Arbeitsspeichernutzung während der Evaluation

Auf der x-Achse ist das Datum dargestellt. Zwei vertikale Linien der Untergrundlinien ergeben einen Tag Ausführungszeit. Es ist ein charakteristisches Muster für das Laden des Eingabe-Graphen, beginnend mit einem linearen Anstieg, zu erkennen. Die langen horizontalen Linien im Verlauf des Graphen sind die Basic Graph Pattern Evaluierungen. Diese sind im gezeigten Ausschnitt eher unterdurchschnittlich lang. Die meisten anderen Features, außer Basic Graph Pattern, sind wie oben bereits, gezeigt mit der Abtastrate von 5 min. nicht sichtbar, da selbst alle Ausführungen des Features zusammen über alle Queries unter 5 min benötigt haben. Die roten Pfeile zeigen den Zeitpunkt zu dem der Datenerfassungs-Rekord, am Ende jeder Query Ausführung, mit entsprechenden „unique_name“ erstellt wurde. Auf diese Weise lassen sich die einzelnen Peaks zu den Wikidata Example Queries zuordnen. Für Queries mit Fehler bei der Übersetzung wurde dies als „Error“ in der Grafik erwähnt. Es ist zu sehen, dass der zweite Error, bei der Query mit Hash „3a780“, nicht sofort nach der vorherigen Query auftrat. Dies liegt daran, dass für diese Query die Evaluierung in Blazegraph etwas mehr Zeit in Anspruch genommen hat. Alle Queries wurden zunächst auf Blazegraph und dann in Nemo ausgeführt. Am Ende der Query mit dem hohen Peak in der Arbeitsspeichernutzung existiert kein roter Pfeil und kein Datenerfassungs-Rekord. Dies liegt daran, dass das Programm wegen zu hoher Speicherauslastung vorzeitig abgebrochen und neu gestartet wurde. Dennoch kann aufgrund der bekannten Reihenfolge der Queries festgestellt werden, dass es sich bei dieser um die Query „Computer Science and Technology/Return a bubble chart of mediatypes by count of file formats“ mit dem unique_name „computer-science-and-2ae07“ handelt. Eine naheliegende Vermutung ist, dass der enorme Speicherbedarf durch den in dieser

Query verwendeten Property Path verursacht wird, welcher alle Dinge findet, deren Typ ein File Format oder eine Subklasse von File Format ist. Da in keiner erfolgreichen Query der Speicherbedarf eines Prädikats höher als der für den Eingabe-Graphen ist[3] und im Peak der Speicherbedarf andere Queries über das Doppelte übersteigt, ist es sehr wahrscheinlich, dass der hohe Speicherbedarf durch temporären Zwischenergebnisse verursacht wird. Genaue Aussagen sind allerdings generell sehr schwierig, da das manuelle debuggen von Wikidata Queries in Nemo durch die dreistündige Wartezeit beim Importieren der Daten, sehr erschwert wird.

Zusammenfassend ist festzuhalten, dass einige Queries die Grenzen des Speichers oder der akzeptablen Verarbeitungszeit überschreiten. Dies könnte in einer anderen Nemo Version unter Umständen durch folgende Ansätze verbessert werden:

- Optimierung der Import Zeit, z.B. durch ein Feature, welches es erlaubt, Daten einmalig zu importieren und für mehrere Nemo-Programme zu verwenden. Dies würde die Gesamtausführungszeit deutlich reduzieren und das Debuggen deutlich vereinfachen.
- Optimierung der Regel-Körper Evaluierung, z.B. durch stärkere Berücksichtigung von Konstanten. Basic Graph Patterns können häufig durch Einschränken der betrachteten Daten durch konstante IRIs in Prädikat Position effizienter evaluiert werden. Die durchgeführte Evaluierung bietet Evidenz, dass dies in der verwendeten Version von Nemo häufig noch nicht geschieht.
- Optimierung temporärer Zwischenergebnisse, z.B. durch Ausführung von Regeln in kleineren Schritten. Dies könnte in vielen Fällen verhindern, dass ein Datalog-Programm aufgrund des Speicherlimits nicht ausgeführt werden kann.

Es kann in der Evaluierung gezeigt werden, dass in zahlreichen Fällen der Ansatz in dieser Arbeit selbst bis zu über 3 Miliarden Tripels skaliert.

[3] die Query, welche nicht vom Eingabe-Graphen abhängt, bildet die einzige Ausnahme, da Nemo den Eingabe-Graphen erst lädt, wenn dieser benötigt wird

5 Zusammenfassung und Ausblick

In dieser Arbeit konnte gezeigt werden, wie sich fast alle SPARQL-Query Features zu Nemo übersetzen lassen. Als besonders herausfordernd erwies sich dabei der SPARQL konforme Umgang mit nicht gebundenen Variablen, wie etwa durch ein `OPTIONAL` in SPARQL. Die Übersetzung wurde konkret implementiert. Um die zahlreichen Features in der gegebenen Zeit implementieren zu können, wurde eine Nemo Template Language entwickelt, welche verallgemeinerte Nemo Regeln mittels Rust-Makros erzeugen kann. Durch eine Evaluation konnte gezeigt werden, dass die Übersetzung in weiten Teilen SPARQL konform ist und für viele Queries bis zu Milliarden von Tripels im Eingabegraphen skaliert.

Die Übersetzung fand grundsätzlich bottom up entlang der SPARQL-Algebra Baumstruktur statt. Dieser Ansatz zeichnete sich durch seine Einfachheit aus, erlaubte jedoch nicht eine vollständig SPARQL konforme Implementierung des Exists filters, da durch diesen Expressions auch von Variablen aus Operationen, welche im SPARQL-Algebra-Baum oberhalb angeordnet sind, abhängen können.

Es wäre interessant, Ansätze zu betrachten, durch welche Zwischenergebnisse nicht nur in übergeordneten Operationen im SPARQL Algebra Baum beachtet werden. Auf diese Weise könnte der Exists filter korrekt implementiert werden. Außerdem ist dadurch ein Effizienzgewinn zu erwarten, da die Größe der Zwischenergebnisse durch die zusätzlichen Einschränkungen reduziert werden kann. Dies ist insbesondere relevant für Property Paths, welche zu großen Zwischenergebnissen neigen.

Eine weitere Optimierung könnte betrachtet werden, mit welcher Regeln aus dem Nemo-Programm entfernt werden, indem diese in andere Regeln direkt eingebettet werden. Ist beispielsweise ein Prädikat *p* durch die Prädikate *u* und *v* definiert, könnte in einer Regel, welche das Prädikat *p* im *Regel-Körper*↗ hat, dieses in einigen Fällen durch die Prädikate *u* und *v* ersetzt werden. Dadurch wird Nemo mehr Freiraum für Optimierungen gegeben. Eine umfangreiche Implementierung dieser Optimierung könnte auch Expressions in Nemo berücksichtigen. Dies kann einen weiteren Geschwindigkeitsgewinn erbringen, da in Nemo typischerweise eine zusammengesetzte Expression effizienter ist als die jeweiligen Teilepressions in einzelnen Regeln.

Eine direkte Anwendung der Übersetzung von SPARQL-Queries nach Nemo könnte auch das Generieren von Teilen einer Test-Suite zum Optimieren von Nemo sein. Beispielsweise bilden die Nemo-Programme der übersetzten Wikidata Beispielqueries, wie in der Evaluation dieser Arbeit beschrieben, zahlreiche realitätsnahe Testfälle mit hoher Komplexität. Dabei ist zu beachten, dass zyklische Abhängigkeiten von Prädikaten mit großer Zyklen Länge in den aus SPARQL-Queries resultierenden Nemo Programmen nicht vorkommen.

Literatur

[1] Renzo Angles und Claudio Gutierrez. „The expressive power of SPARQL“. In: *International Semantic Web Conference*. Springer. 2008, S. 114–129.

[2] Renzo Angles u. a. „SparqLog: A System for Efficient Evaluation of SPARQL 1.1 Queries via Datalog [Experiment, Analysis and Benchmark]“. In: *arXiv preprint arXiv:2307.06119* (2023).

[3] Stefano Ceri, Georg Gottlob, Letizia Tanca u. a. „What you always wanted to know about Datalog (and never dared to ask)“. In: *IEEE transactions on knowledge and data engineering* 1.1 (1989), S. 146–166.

[4] Alex Ivliev u. a. „Nemo: Your Friendly and Versatile Rule Reasoning Toolkit“. In: *Proceedings of the 21st International Conference on Principles of Knowledge Representation and Reasoning (KR 2024)*. Hrsg. von Pierre Marquis, Magdalena Ortiz und Maurice Pagnucco. Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning. IJCAI Organization, Nov. 2024, S. 743–754. DOI: https://doi.org/10.24963/kr.2024/70.

[5] *rdflib 7.1.2*. URL: https://rdflib.readthedocs.io/en/stable/ (besucht am 17. 01. 2025).

[6] Simon Schenk. „A sparql semantics based on datalog“. In: *Annual Conference on Artificial Intelligence*. Springer. 2007, S. 160–174.

[7] *SPARQL 1.1 Query Language*. URL: https://www.w3.org/TR/sparql11-query/ (besucht am 19. 12. 2024).

[8] *SPARQL 1.1 Test Suite*. URL: https://www.w3.org/2009/sparql/docs/tests/summary.html (besucht am 19. 12. 2024).

6 Anhang

6.1 Begriffsverzeichnis

- *Ergebnis* ↗ – Abschnitt 1.5 (S.13)
- *gebunden* ↗ – Abschnitt 1.5 (S.13)
- *Lösung* ↗ – Abschnitt 1.5 (S.13)
- *Query-Ergebnis* ↗ – Abschnitt 1.4 (S.12)
- *Regel-Kopf* ↗ – Abschnitt 1.5 (S.13)
- *Regel-Körper* ↗ – Abschnitt 1.5 (S.13)
- *repräsentiert* ↗ – Abschnitt 1.5 (S.13)
- *steht für* ↗ – Abschnitt 1.5 (S.13)
- *Triple-Graph* ↗ – Abschnitt 1.3 (S.11)
- *UNDEF-Marker* ↗ – Abschnitt 3.1 (S.35)
- *zum Tragen* ↗ kommen – Abschnitt 1.5 (S.13)

6.2 Abkürzungsverzeichnis

- IRI – Internationalized Resource Identifier
- URL – Uniform Resource Locator
- w3c – World Wide Web Consortium
- SPARQL – SPARQL Protocol and RDF Query Language
- SQL – Structured Query Language
- RDF – Resource Description Framework
- BNode – Blank Node
- stdin – standard input
- stdout – standard output

6.3 Diskussion über Implementierung zum Tracken von Attribut-Eigenschaften in der Nemo Template Language

Sei $a(?x) \iff b(?x) \ \wedge \ c(?x)$ eine Regel und p ein Prädikat in der Definition der Eigenschaft. Es wird angenommen, dass dies die einzige Regel für a im Datalog Programm ist. Für diesen Spezialfall kann die Regel als logische Äquivalenz gesehen werden.

geg.: „Die Eigenschaft gilt für b **oder** c“ $\Box\forall ?x : (b(?x) \Rightarrow p(?x)) \vee \Box\forall ?x : (c(?x) \Rightarrow p(?x))$

zu zeigen: $\Box\forall ?x : (a(?x) \Rightarrow p(?x))$

Die zu zeigende Aussage ist durch Regel äquivalent zu: $\Box\forall ?x : (b(?x) \wedge c(?x) \Rightarrow p(?x))$

If the reasoner said the property is true, then it is really true:

$$\Box\forall ?x : (b(?x) \Rightarrow p(?x)) \vee \Box\forall ?x : (c(?x) \Rightarrow p(?x))$$

stronger condition preserves implication

e.g. $b(?x) \wedge c(?x) \implies b(?x)$

$$\implies \Box\forall ?x : (b(?x) \wedge c(?x) \Rightarrow p(?x)) \vee \Box\forall ?x : (b(?x) \wedge c(?x) \Rightarrow p(?x))$$

$(x \vee x) = x$

$$\implies \Box\forall ?x : (b(?x) \wedge c(?x) \Rightarrow p(?x))$$

If the reasoner said the property is false, then it may not really be false:

Standard Umformungen, Verschieben der Negation:

$$\neg(\Box\forall ?x : (b(?x) \implies p(?x)) \vee \Box\forall ?x : (c(?x) \implies p(?x)))$$
$$\neg\Box\forall ?x : (b(?x) \implies p(?x)) \wedge \neg\Box\forall ?x : (c(?x) \implies p(?x))$$
$$\Diamond\exists ?x : \neg(b(?x) \implies p(?x)) \wedge \Diamond\exists ?x : \neg(c(?x) \implies p(?x))$$

Standard Umformungen, Auflösen der Implikation:

$$\Diamond\exists ?x : \neg(b(?x) \implies p(?x)) \wedge \Diamond\exists ?x : \neg(c(?x) \implies p(?x))$$
$$\Diamond\exists ?x : \neg(\neg b(?x) \vee p(?x)) \wedge \Diamond\exists ?x : \neg(\neg c(?x) \vee p(?x))$$
$$(\Diamond\exists ?x : b(?x) \wedge \neg p(?x)) \wedge (\Diamond\exists ?x : c(?x) \wedge \neg p(?x))$$

… Show or assume that $b(?x)$ and $c(?x)$ may not contradict, e.g., if $b = c$ . . . Diese Umformung ist fragwürdig, siehe Argumentation unten.

Standard Umformungen, Bilden der Implikation:

$$\Diamond\exists ?x : (b(?x) \wedge c(?x) \wedge \neg p(?x))$$
$$\Diamond\exists ?x : \neg(\neg b(?x) \vee \neg c(?x) \vee p(?x))$$
$$\Diamond\exists ?x : \neg(\neg(b(?x) \wedge c(?x)) \vee p(?x))$$

Standard Umformungen, Verschieben der Negation

$$\Diamond\exists ?x : \neg(b(?x) \wedge c(?x) \implies p(?x))$$
$$\neg(\Box\forall ?x : (b(?x) \wedge c(?x) \implies p(?x)))$$

Argumentation mittlerer Schritt:

In Standard-Modallogik ist die Umformung falsch. Ein Gegenbeispiel existiert mit $b(?x) \iff \neg c(?x)$ (für alle $?x$), was grundsätzlich konsistent mit der Prämisse ist, aber $b(?x) \wedge c(?x)$ notwendigerweise falsch macht.

Auf der anderen Seite wäre es ohne weitere Information *möglich*, dass $b(?x) \iff c(?x)$ (für alle $?x$) gilt. Ersetzt man c durch b, ist leicht zu sehen, dass die Ausdrücke identisch sind.

In der Praxis kann man sagen, dass die Inkorrrektheit der Schlussfolgerung nach Standard-Modallogik ohne weitere Annahmen dazu führt, dass in einigen Fällen eine Optimierung nicht durchgeführt werden kann, da durch die konservative Abschätzung das Eintreten eines Falles nicht ausgeschlossen werden kann, obwohl dies praktisch nicht passiert. Außerdem existiert in der Praxis ein Kontinuum zwischen $b(?x) \iff \neg c(?x)$ und $b(?x) \iff c(?x)$. Im Allgemeinen ist die Wahrscheinlichkeit für die Korrektheit der Schlussfolgerung höher, je häufiger sowohl $b(?x)$ als auch $c(?x)$, für $?x$ gilt.