Repräsentation der Datenmodelle von Graphdatenbanken als formale Graphen

v1.0

Thomas Fuhr

thomas.fuhr@th-nuernberg.de

https://orcid.org/0000-0002-9935-7848

Technischer Bericht

Fakultät Informatik

Zusammenfassung

Aufgrund der zunehmenden Beliebtheit von Graphdatenbanken gewinnt die Frage ihrer Interoperabilität zunehmend an Bedeutung. Während das Datenmodell von RDF-Graphdatenbanken standardisiert ist, gilt dies nicht für Labeled-Property-Graphdatenbanken. Ein genauerer Blick zeigt, dass sowohl in der Literatur wie auch von Systemanbietern der Begriff Labeled Property-Graph im Detail unterschiedlich verstanden wird. Um Brücken zwischen den Datenmodellen zu schaffen, ist die Kenntnis der zugrunde gelegten Graphformalisierungen wie ein genaueres Verständnis konkreter Transformationen zwischen den verschiedenen Graphrepräsentationen erforderlich. Dieser Bericht erfasst einige relevante Graphdefinitionen i. S. der mathematischen Graphentheorie, welche als Grundlage formaler Beschreibungen verwendeter Datenmodelle dienen können. Basierend auf einer weit gefassten Definition für Labeled-Property-Graphen wird eine Typ-Notation eingeführt, um die in Literatur und Graphdatenbanksystemen vorhandenen unterschiedlichen Ausprägungen des Begriffs charakterisieren und strukturieren zu können. Im Weiteren werden unterschiedliche Transformationen von RDF-Graphen in formale Graphen, insbesondere Labeled-Property-Graphen, einheitlich dargestellt und an einem durchgängigen Beispiel illustriert.

Inhalt

Tabellen

Abbildungen

1 Einführung

Während das Management großer Datenbestände jahrzehntelang durch den Einsatz immer ausgefeilterer Datenbanksysteme auf Basis des relationalen Datenmodells zufriedenstellend gelöst werden konnte, haben sich in den letzten ca. 10 Jahren ergänzend sog. NoSQL[1]-Datenbanksysteme etabliert. Diese Gruppe von Datenmanagementsystemen adressiert Herausforderungen, welche sich durch die weiterhin immens steigenden Datenmengen (Big Data), spezifische Datenbestände (z.B. Dokumente), andere Datennutzungsprofile (Analytical Processing statt Transactional Processing) oder sich zunehmend schneller anzupassende Anwendungssysteme ergeben. Durch die Verfügbarkeit dieser Datenbanksysteme, welche andere Datenmodelle zugrunde legen, können für die vorliegenden oder zu erhebenden Daten spezifisch zugeschnittene Datenmanagementlösungen entwickelt bzw. eingesetzt werden.[2]

Die Webseite db-engines.com verfolgt seit 2013 die Entwicklung der Popularität von Datenbanksystemen. Zur Messung des Popularitätswerts werden unter anderem die Häufigkeit der Erwähnung der Systeme auf Webseiten, in Tweets, Stellenanzeigen, beruflichen Netzwerken und fachlichen Q&A-Foren berücksichtigt, wie auch die Häufigkeit von relevanten Google Trends-Anfragen[3] . Im November 2013 wurden 77 relationale und 127 NoSQL Datenbanksysteme erfasst. Ein Blog-Post aus diesem Zeitraum stellt fest, dass die relationalen Systeme zwar klar beherrschend sind, dass aber NoSQL-Datenbanken sichtbar an Beliebtheit zunehmen[4]. Diese Analyse hat sich bestätigt. Lag der in 2013 von db-engines.com für insgesamt 204 betrachtete Systeme ermittelte Popularity Score noch bei 90,8% für relationale Datenbanksysteme gegenüber nur knapp 10% für die betrachteten NoSQL-Systeme, wurden im August 2021 Popularity Scores von 72,8% vs. 37,2% ausgewiesen. Zudem hat auch gegenüber 2013 die Vielfalt der genutzten Datenbanksysteme weiter zugenommen. Die jeweils aktuellen Zahlen sind unter db-engines.com/en/ranking_categories zu finden.

Graphdatenbanken (kurz: *GDB*) sind eine spezifische Gruppe von NoSQL-Datenbanken, welche erlauben Datenbestände in Form von Graphen zu repräsentieren und zu managen. Sie unterstützen i.d.R. die für Datenbanken üblichen CRUD[5]-Operationen und OLTP[6].

Für einen groben Überblick werden Graphdatenbanken oft nach zwei (orthogonalen) Gesichtspunkten kategorisiert: der genutzten Speicherstruktur bzw. der Art des verwendeten Graphmodells. Wird eine spezifisch entwickelte Speicherstruktur verwendet, d.h. ein sog. *native graph storage*, spricht man auch von *nativen Graphdatenbanken*. Nicht-native Graphdatenbanken nutzen dagegen klassische relationale oder andere NoSQL-Datenbanken zur Speicherung ihrer Graphen. Hinsichtlich des Graphmodells haben sich zwei Hauptkategorien herausgebildet: *RDF-Graphen*, welche auf dem *Resource Description Framework* (kurz: *RDF*) basieren, sowie Nicht-RDF-Graphdatenbanken. Letztere werden mit Blick auf das von ihnen überwiegend verwendete Graphmodell *Labeled-Property-Graphdatenbanken* (kurz: *LP-GDB*) oder oft auch *Property-Graphdatenbanken* (kurz: *P-GDB*) genannt. Hieraus folgt eine Einteilung der Graphdatenbanken in *RDF-Graphdatenbanken* und *Labeled-*

[1] *N*ot only *SQL*
[2] Einen guten Überblick über NoSQL-Systeme bieten z.B. Gessert et al. (2017), Davoudian et al. (2018). Verweise auf zahlreiche Systeme sind z.B. unter nosql-database.org zu finden.
[3] siehe db-engines.com/en/ranking_definition
[4] siehe db-engines.com/de/blog_post/23
[5] Create, Read, Update, Delete
[6] Online Transactional Processing

Property-Graphdatenbanken.

Für eine detailliertere taxonomische Gliederung wird der Leser auf die exzellente Übersichtsarbeit von Besta et al. (2020) verwiesen. Auf Basis der von ihnen ausgearbeiteten Taxonomie charakterisieren sie 45 aktuell verfügbare Graphdatenbanksysteme[7]. Zudem skizzieren Besta et al. in ihrer Arbeit an ausgewählten Systemen wie diese ihre Graphen nativ oder auch nicht-nativ speichern (Abschnitte 2.3 sowie 4.). Eine Differenzierung zugrunde liegender formaler Graphmodelle ist in Bonifati et al. (2018), Kap. 2, ausgearbeitet.

Gemäß db-engines.com/en/ranking_categories (Stand August 2021) haben alle Graphdatenbanken[8] zusammengenommen mit 2,1% nur einen kleinen Popularity Score mit Blick auf alle Datenbanktypen. Innerhalb der Gruppe der NoSQL-Datenbanken sind sie jedoch mit 7,7% inzwischen der fünft beliebteste Datenbanktyp[9]. Zudem zeigen sie seit 2013 die stärkste Popularitätssteigerung. 41 der 373 betrachteten verschiedenen DB-Systeme unterstützen ein Graphmodell. Bemerkenswert ist dabei, dass die für die Nutzung des relationalen Modells beliebtesten kommerziellen Systeme - Oracle, Microsoft SQL Server und IBM Db2 - inzwischen auch ein Graphmodell als weitere Option zur Datenrepräsentation anbieten[10]. Graphdatenbanken bilden somit eine Klasse von NoSQL-Datenbanken, welche sich in den letzten Jahren fest etabliert hat.

Ein Grund hierfür ist der nutzbringende Einsatz von Graphdatenbanken zur Lösung interessanter und auch wirtschaftlich relevanter Problemstellungen. Für diese Anwendungen ist es z.B. von Interesse das Beziehungsgeflecht zwischen den relevanten Objekten einer Domäne in besonderer Weise auswerten zu können. Dies kann der direkte Kontext der Objekte sein, die Analyse von Beziehungsketten, der Vergleich von Beziehungsmustern verschiedener Objekte u.a. Die Repräsentation der Daten als Graph(en) bietet hierfür eine ideale Grundlage. In Form sog. *Knowledge Graphs* werden oft unterschiedlichste Datenquellen zusammengeführt. Anwendungen sind breit gestreut und finden sich z.B. in Verwaltung, Wirtschaft, Industrie, Sozialen Netzwerken, im medizinischen Sektor wie im Bereich des Militärs. Sie reichen von der Aufdeckung betrügerischen Verhaltens z.B. im Steuer- und Finanzwesen, über die Analyse von Störungsquellen und -auswirkungen in Supply-Chains, Fertigungsketten oder IT-Netzen, über die Sichtbarmachung von Wirkungszusammenhängen pharmazeutischer Stoffe, zu Master Data Management, 360-Grad-Kundenanalysen sowie Empfehlungssystemen im Handel. Beschreibungen zu typischen Anwendungsfeldern sind z.B. in Robinson et al. (2015), Kap. 5 oder Blumauer (2020), S. 55 ff. zu finden. Weitere Skizzen von Anwendungsbereichen, Use Cases und Hinweise auf konkrete Nutzer GDB-basierter Systeme sind zahlreich in Publikationen und auf Webseiten der Anbieter von Graphdatenbanksystemen zu finden[11]. Für Herstellerlisten siehe auch die von Besta et al. (2020) verwendeten Web-Quellen unter Fußnote [7].

Mit zunehmender Anzahl und Lebensdauer von verwalteten Datenbeständen entstehen Fragen hinsichtlich der Interoperabilität, welche sich durch veränderte Organisationsstrukturen eines Unternehmens, die Einführung neuer Technologien oder andere Gründe ergeben. Z.B. (siehe auch Thakkar (2021), S. 3):

- Wie können (Teil)Datenbestände von einem System zu einem anderen migriert werden?
- Können unterschiedliche Anwendungen Datenbestände gemeinsam nutzen?

[7] Die Auswahl dieser Systeme basiert auf den Webseiten db-engines.com/en/ranking/graph+dbms, nosql-database.org, database.guide, www.g2crowd.com/categories/graph-databases sowie www.predictiveanalyticstoday.com/top-graph-databases

[8] Anders als in unserem Bericht, werden auf db-engines.com RDF-GDB-Systeme als *RDF Stores* bezeichnet und Labeled Property-GDB als *Graph DBMS*.

[9] nach Document Stores (36,9%), Key-Value Stores (19,9%), Search Engines (17%) und Wide Columns Stores (11,1%)

[10] siehe `https://db-engines.com/en/ranking/relational+dbms` und db-engines.com/en/ranking_osvsc

[11] siehe z.B. ORACLE (2021), neo4j.com/use-cases/, vaticle.com/, ontotext.com/knowledge-hub/, stardog.com/use-cases/, tigergraph.com

- Wie können verschiedene Datenbestände zusammengefasst werden?
- Ist es möglich eine Integrationsschicht über Datenbestände in verschiedenen Systemen zu bilden?

Dies ist unabhängig davon, ob die unterschiedlichen Datenbestände in Datenbanken desselben Typs (z.B. ausschließlich relationales Modell) gehalten werden, oder, ob Systeme mit unterschiedlichen Datenmodellen zu ihrer Verwaltung genutzt werden.

Die Entwicklung von Standards ist ein üblicher Weg, um Interoperabilität zu unterstützen. Solche Standards zu entwickeln ist i. allg. innerhalb von Systemen, welche das gleiche Datenmodell verwenden, einfacher möglich. Die Überbrückung der Unterschiede verschiedener Datenmodelle ist dagegen eine deutlich komplexere Herausforderung und ggf. auch gar nicht realisierbar.

Für relationale Datenbanken, welche alle auf der Relationenalgebra und der Organisation der Daten in Tabellen basieren, war die Einführung des SQL[12]-Standards ISO/IEC CD 9075 sicher ein wesentlicher Fortschritt mit Blick auf Interoperabilität. Gleichwohl zeigte sich auch, dass die Harmonisierung der Managementsprache allein nicht alle Problemstellungen bzgl. der Interoperabilität lösen kann. Das betrifft vor allem die semantischen Aspekte, denn die Modellierung der Repräsentation von Datenbeständen ist kein standardisierter Prozess.

RDF-GDB und Labeled Property-GDB unterscheiden sich deutlich hinsichtlich des Stands der Standardisierung. Für RDF-GDB existieren zahlreiche sog. *Recommendations* des World Wide Web Consortiums (W3C), insbesondere für das Graphmodell RDF (Cyganiak et al. 2014), die Managementsprache SPARQL (Harris and Seaborne 2013) und verschiedene Serialisierungsformate zur textuellen Repräsentation eines Datenbestands, wie z.B. RDF/XML (Gandon and Schreiber 2014), N-Triples (Beckett 2014) oder Turtle (Beckett et al. 2014). Für Labeled Property-GDB existieren solche Standards bisher nicht. Es existieren z.B. zahlreiche Managementsprachen wie Cypher, OpenCypher, Gremlin, GraphQL, PGQL u.a.[13]. Allerdings gibt es Aktivitäten zur Erarbeitung von Standards. So wurde 2019 ein offizielles Projekt auf den Weg gebracht, um eine ISO-standardisierte Managementsprache für Labeled-Property-GDB zu entwickeln, die sog. *Graph Query Language* kurz: *GQL* [14]. Damals wurde als ambitioniertes Zeitziel August 2022 angegeben. Betrachtet man die aktuellen Inhalte der GQL-Website, so scheint der Prozess jedoch aktuell zu stocken. Als eine weitere Aktivität hat sich 2018 informell die *Property Graph Schema Working Group (PGSWG)* gebildet, welche Empfehlungen für eine Standardisierung von Labeled Property-Graph-Schemata erarbeiten möchte (siehe Positionspapier *Property Graphs need a Schema*[15]).

Das Thema, wie Brücken zwischen beiden GDB-Ansätzen aufgebaut werden können, beschäftigt die Graphdatenbanken-Community zunehmend. Im März 2019 fand ein vom W3C organisierter *Workshop on Web Standardization for Graph Data Creating Bridges: RDF, Property Graph and SQL*[16] mit Beteiligung von Vertretern beider GDB-Typen, sowohl von Systemanbietern wie auch dem akademischen Kontext, statt. Die Tagung SEMANTiCS 2021 widmete sich ebenfalls der Problematik der Interoperabilität zwischen RDF- und Property-Graphen mit dem "'1st Workshop on Squaring the Circle on Graphs"'.

Mit Blick auf die Interoperabilität von Graphdatenbanken, sind insb. drei Arten von Interoperabilität von Interesse: *syntaktische*, *semantische* und *Query-Interoperabilität*. Syntaktische Interoperabilität betrifft die Frage gemeinsamer oder ineinander transformierbarer Serialisierungen zum Austausch des Datenbankinhalts. Semantische Interoperabilität drückt aus, dass beim Austausch der Graphdaten

[12] *S*tandard *Q*uery *L*anguage
[13] gqlstandards.org/existing-languages
[14] siehe The GQL Manifesto sowie gqlstandards.org
[15] https://www.w3.org/Data/events/data-ws-2019/assets/position/Juan%20Sequeda.txt
[16] www.w3.org/Data/events/data-ws-2019/

deren Bedeutung erhalten bleibt. Dies schließt den Aspekt mit ein, dass vorhandene implizite oder explizite Schemata mit ausgetauscht werden können. Query Interoperabilität bedeutet, dass eine Übersetzung von Queries zwischen den Query Languages verschiedener Systeme möglich ist. (siehe Heiler (1995), Angles et al. (2019) und Corcho et al. (2021))

Alle drei Arten der Interoperabilität haben gemein, dass zu ihrer Umsetzung zunächst ein genaues Verständnis der zugrunde liegenden Datenmodelle vorhanden sein muss. Ein Schritt zu diesem Verständnis ist, zu klären, welche Art von formalen Graphen, damit meinen wir Graphen i. S. der mathematischen Graphentheorie, von RDF- und Labeled Property-GDB repräsentiert werden. Auf Basis dieser Erkenntnisse können Graphtransformationen beschrieben und untersucht werden.

Ein genauerer Blick zeigt, dass RDF-Graphen seitens des W3C nicht als formale Graphen im Sinne der math. Graphentheorie beschrieben sind. Es fehlt hierfür an einheitlichen Festlegungen. Auch Labeled-Property-Graphen werden oft informell beschrieben (siehe Robinson et al. (2015)) und sind in ihren Charakteristika nicht standardisiert. Hier ist es aus unserer Sicht wertvoll, die Unterschiede der verwendeten Graphen einheitlich beschreibbar zu machen.

Gegenstand dieses Berichts ist es, unterschiedliche in der Literatur präsentierte Ansätze der Repräsentation von RDF- und Labeled-Property-Graphen als formale Graphen gemeinsam auf Basis einer einheitlichen Formalisierung zu beschreiben. Hiermit soll die Gegenüberstellung der verschiedenen Repräsentationsvorschläge unterstützt werden. Für die vorkommenden relevanten Hauptvarianten formaler Graphen - insb. Labeled-Graphen und Labeled-Property-Graphen - führen wir jeweils eine Typ-Notation ein. Diese erlaubt unterschiedliche Ausprägungen innerhalb der Hauptvarianten prägnant zu charakterisieren. Dies ist von Vorteil, da sich zeigt, dass sowohl in der Literatur vorzufindende Definitionen von Labeled-Property-Graphen wie auch die von Labeled-Property-Graphdatenbanksystemen verwendeten Graphmodelle im Detail voneinander abweichen.

Ansätze zur Repräsentation von RDF-Graphen als Labeled-Property-Graphen werden oft als *Transformationen* bezeichnet. Dies betont den Beitrag dieser Ansätze hinsichtlich der Interoperabilität zwischen den beiden Graphdatenmodellen. Da RDF-Graphen nicht als formale Graphen definiert sind, kann darüberhinaus jede Variante ihrer Formalisierung als Graph als Ergebnis einer Transformation verstanden werden. In diesem Bericht wird jede Transformation schematisch als Tabelle, durch eine Definition des Ergebnisgraphen (wir sprechen auch vom zugeordneten formalen Graphen) und anhand eines Beispiels beschrieben. Ferner ordnen wir die zugeordneten Graphen in unser Typ-Schema ein. Hieraus wiederum können Hinweise gewonnen werden, in welchen konkreten GDB-Systemen, die Transformationsergebnisse repräsentierbar sind.

Dieser Bericht spiegelt *work in progress* wieder. Zahlreiche Aspekte hinsichtlich der Formalisierung von RDF-Graphen finden noch keine Berücksichtigung[17]. Zudem werden die für die Interoperabilität der Graphdatenbanksysteme ebenso wichtigen und interessanten Transformationen von Labeled-Property-Graphen in RDF-Graphen nicht behandelt.

In Kap.2 fassen wir wichtige Begriffe des Resource Description Frameworks zusammen. Dieses Kapitel enthält auch einen Beispiel-RDF-Graphen, welcher der Illustration der betrachteten Transformationen dient. Im anschließenden Kap.3 definieren wir verschiedene Arten formaler Graphen, welche als Grundlage für den weiteren Bericht relevant sind, und führen eine Typ-Notation ein, welche wesentliche Differenzierungen zu beschreiben erlaubt. Welche Hinweise sich aus den RDF-spezifischen Recommendations für die Formalisierung von RDF-Graphen ergeben, ist Gegenstand des Kap. 4. Schließlich werden in den Kap. 5, 6 und 7 verschiedene Transformationen von RDF-Graphen vorgestellt und auf unseren Beispiel-RDF-Graphen angewendet. Aus Layout-Gründen werden Ergebnisgraphen dieser Transformationen z. T. am Ende der jeweiligen Kapitel dargestellt. Der Bericht endet mit zusammenfassenden und ausblickenden Bemerkungen in Kap. 8.

[17] siehe auch Kap. 8

2 RDF - Grundlegende Begriffe

Zunächst fassen wir für diesen Bericht relevante Begriffe des Resource Description Frameworks (RDF) zusammen.

Das RDF-Datenmodell baut auf *RDF-Tripeln* der Form (*Subjekt,Prädikat,Objekt*) auf, mit denen Aussagen, sog. *RDF Statements*, über *Ressourcen* einer Wissensdomäne getroffen werden können. Dabei gilt *" Anything can be a resource, including physical things, documents, abstract concepts, numbers and strings "* (Cyganiak et al. (2014), Abschnitt 1.1). RDF-Tripel formalisieren damit Aussagen wie z.B. "John liebt Kitty" oder "John heißt mit Nachnamen 'Doe'".

Die formalen Darstellungen der Ressourcen heißen *RDF-Terme*. Man unterscheidet drei Arten von RDF-Termen, welche als zueinander disjunkt betrachtet werden. Ein RDF-Term des Typs *IRI* repräsentiert eine global eindeutige Ressource. Jeder IRI-Term ist syntaktisch ein Internationalized Resource Identifier[1]. Ein *Blank Node* repräsentiert eine unbenamte Ressource, für die eine globale Identifizierung nicht sinnvoll oder notwendig ist und es ausreicht, sie ausschließlich durch über sie gemachte RDF Statements zu charakterisieren. Zum Zweck ihrer textuellen Beschreibung (in Serialisierungsformaten) werden Blank Nodes syntaktisch durch lokal gültige Bezeichner identifiziert (siehe *RDF-Datasets*, weiter unten). IRIs und Blank Nodes repräsentieren (komplexere) Objekte der Domäne, welche durch Statements attributiert oder untereinander in Beziehung gesetzt werden können. Ein *Literal* repräsentiert eine Ressource, welche der Wert eines Attributs ist. Ein Literal kann syntaktisch einfach durch einen String gegeben sein (einfaches Literal). Die Interpretation des dadurch ausgedrückten verbundenen Attributwerts ist in diesem Fall intuitiv. Ergänzend kann im Literal zusätzlich zum String ein Datentyp angegeben werden, durch welchen die Interpretation des Strings explizit spezifiziert wird. Üblich ist die Verwendung der XML-Datentypen (Peterson et al. 2012). Zur Unterstützung der Mehrsprachigkeit können Literale alternativ ein Sprach-Tag enthalten. Ist dies der Fall, so ist der Literal-String obligatorisch vom Datentyp `http://www.w3.org/1999/02/22-rdf-syntax-ns#langString`. Für Literale mit Sprach-Tag wird deshalb der Datentyp in den Serialisierungsformaten i. d. R. weggelassen.

Ein RDF-Tripel (s, p, o) steht für eine Aussage über die durch das Subjekt s repräsentierte Ressource. Dabei gelten für die Tripelbildung folgende Restriktionen: Als Subjekte dürfen lediglich IRIs oder Blank Nodes, als Prädikate ausschließlich IRIs und als Objekte alle drei Arten von RDF-Termen verwendet werden. Ist das Tripel-Objekt o ein Literal, dann wird dem Subjekt s der Attributwert o des Attributs p zugeordnet. Ist das Tripel-Objekt o kein Literal, dann repräsentiert das Prädikat p eine binäre gerichtete Relation zwischen Domänenobjekten und es gilt: s steht in der Relation p zu o. Das Prädikat p ist gemäß RDF-Nomenklatur eine *RDF-Property*. Wichtig ist zu erkennen: Während s und o für *konkrete Ausprägungen* von Domänenobjekten stehen, ist dies für p *nicht* der Fall. p steht für den *Typ* des/der hier verwendeten Attributs/Relation. Das Tripel (s, p, o) beschreibt eine Instanz dieser Relation.[2]. Der OWL Recommendation (Group 2012) folgend bezeichnen wir RDF-Propertys, welche Attribute repräsentieren, als *Datatype Propertys* und jene, die Domänenobjekte in Beziehung setzen, als *Object Propertys*.

Ein *RDF-Graph* ist eine endliche Menge von RDF-Tripeln. RDF 1.1 erlaubt es RDF-Graphen zu benamen und Kollektionen zu bilden. Es gilt gemäß Cyganiak et al. 2014, Abschnitt 4.:

[1] gemäß RFC3987 (https://www.ietf.org/rfc/rfc3987.txt)
[2] Im math. Sinn gilt also $(s, o) \in p$

Listing 2.1: RDF-Graph $R^{Pets-Inst}$: Beschreibung der Instanzen der Beispieldomäne "Pets" (Turtle-Syntax).

```
@prefix gn: <http://www.geonames.org/> .
@prefix exs: <http://example.org/pets/schema/1.0/> .
@prefix exd: <http://example.org/pets/data/1.0/> .

exd:Kitty exs:name "Kitty"^^xsd:string .
exd:Kitty rdf:type exs:Pet .
exd:Kitty rdf:type exs:Cat .
exd:Kitty exs:color "black" .
exd:Kitty exs:hates exd:Bliss .

exd:Bliss exs:name "Bliss"@en .
exd:Bliss rdf:type exs:Dog .
exd:Bliss exs:age "5"^^xsd:integer .
exd:Bliss exs:color "black" .

exd:JD exs:givenName "John"^^xsd:string .
exd:JD exs:familyName "Doe" .
exd:JD exs:pet exd:Kitty .
exd:JD exs:pet exd:Bliss .
exd:JD exs:loves exd:Kitty .
exd:JD exs:loves exd:Bliss .
exd:JD exs:likes exd:Thing99 .
exd:JD exs:livesAt _:genId1 .

_:genId1 rdf:type exs:Address
_:genId1 exs:streetName "Main Road" .
_:genId1 exs:houseNumber "5"^^xsd:integer .
_:genId1 exs:city gn:2633351 .

gn:2633351 gn:officialName "York"@en .
gn:2633351 gn:shortName "York"@en .
gn:2633351 gn:population "208367" .
```

Ein *benannter RDF-Graph* ist ein Paar bestehend aus einer IRI oder einem Blank Node und einem RDF-Graphen. Der Graph wird durch die IRI global bzw. den Blank Node lokal eindeutig benannt. Insb. können die IRI bzw. der Blank Node Subjekte oder Objekte von RDF-Tripeln sein, womit RDF-Statements über bzw. mittels RDF-Graphen möglich sind.
Ein *RDF-Dataset* besteht aus genau einem unbenannten RDF-Graphen, dem sog. *Default-Graph*, und endlich vielen (innerhalb des Datasets eindeutig) benannten RDF-Graphen. Bezeichner von Blank Nodes sind ausschließlich innerhalb eines Datasets gültig.

Um die in diesem Bericht betrachteten verschiedenen Darstellungen von RDF-Graphen als formale Graphen illustrieren zu können, verwenden wir ein kleines Beispiel, welches als gemeinsame Referenz verwendet wird. Listing 2.1 zeigt den RDF-Graphen, welcher die Instanzen (Individuen) unserer Beispiel-Domäne "Pets" ("Tierliebhaber") beschreibt. Wir verwenden zur Serialisierung die Turtle-Syntax (Beckett et al. 2014). Turtle erlaubt Prefix-Definitionen für Namespaces, wodurch die IRIs abgekürzt notiert werden können. Es werden hier keine Turtle-spezifischen zusammenfassenden syntaktischen Konstrukte z.B. für dasselbe Subjekt oder dasselbe Subjekt-Prädikat-Paar verwendet, damit der RDF-Graph deutlich als Triplemenge sichtbar wird.

Informell kurz zusammengefasst beschreibt dieser RDF-Graph, dass eine Ressource exd:John mit

Vornamen John und Nachnamen Doe zwei Haustiere hat[3], die er beide liebt. Das eine ist die Katze exd:Kitty mit Namen Kitty und das andere der Hund exd:Bliss mit Namen Bliss. exd:Kitty ist explizit als Haustier klassifiziert, für exd:Bliss gilt dies nicht. exd:Kitty und exd:Bliss sind beide schwarz. Der Blank-Knoten _:genid1 repräsentiert Johns Wohnadresse, mit Information zu Straße, Hausnummer und Wohnort.

Strukturell enthält der Graph der RDF-Graph $R^{Pet-Inst}$

- alle Arten von RDF-Termen, d.h. IRIs (per Prefix abgekürzt) (z.B: exd:Thing99), Blank Nodes (_:genid1) und Literale (z.B. "Main Road"),
- RDF-Terme unterschiedlicher Vokabulare (siehe Def. 2.5),
- alle Varianten von Literalen: mit Language-Tag und somit implizit vom Typ rdf:langString (z.B. "York"@en), mit Datentypangabe (z.B. "5"^^xsd:integer) sowie einfache Literale ohne beides (z.B. "black")
- Tripel, welche Domänenobjekten Literale zuweisen, deren Prädikate also Datatype Propertys sind (z.B. _:genId1 exs:streetName "Main Road"),
- Tripel, welche Domänenobjekte mit anderen Domänenobjekten verknüpfen, deren Prädikate also Objekt Propertys sind (z.B. exd:JD exs:loves exd:Bliss),
- Tripel, welche durch Verwendung des Prädikats rdf:type, Ressourcen als Instanzen bestimmter Klassen typisieren (z.B. exd:Bliss als Instanz der Klasse exs:Dog),
- eine Ressource, welche explizit mehrere Typisierungen besitzt (exd:Kitty),
- Mehrfachbeziehungen zwischen Ressourcen (z.B. zwischen exd:John und exd:Bliss)

Mit Blick auf die später folgenden Formalisierungen legen wir in den Def. 2.1 bis Def. 2.5 einige Notationen fest.

Definition 2.1.

$\mathcal{IRI}$, $\mathcal{BN}$ und $\mathcal{LIT}$ bezeichnen die paarweise disjunkten Mengen aller RDF-Terme des Typs IRI, Blank Node bzw. Literal.[4]

$\mathfrak{D}_{RDF} \subset \mathcal{IRI}$ bezeichnet die Menge aller IRIs, welche Datentypen für Literale bezeichnen.

Definition 2.2.

Gegeben sei ein RDF-Graph R.

$IRI(R) \subset \mathcal{IRI}$, $BN(R) \subset \mathcal{BN}$ und $LIT(R) \subset \mathcal{LIT}$ bezeichnen die (disjunkten) Mengen der in den Tripeln von R auftretenden IRI-, Blank Node- bzw. Literal-RDF-Terme.

$RDFTerms(R)$ bezeichnet die Menge aller in R auftretenden RDF-Terme.

Definition 2.3. (vgl. Hayes and Gutiérrez (2004), S.4)

Gegeben sei ein RDF-Graph R.

Für ein Tripel $t = (s, p, o) \in R$ definieren wir $subj(t) := s$, $pred(t) := p$, $obj(t) := o$ und bezeichnen $subj(t)$, $pred(t)$ bzw. $obj(t)$ als *Tripel-Subjekt*, *Tripel-Prädikat* bzw. *Tripel-Objekt*.

$SubjT(R)$, $PredT(R)$ und $ObjT(R)$ bezeichnen die Mengen der in den Tripeln aus R auftretenden Tripel-Subjekte, -Prädikate, bzw. -Objekte.

[3] Beachte: "hat Haustiere" (exd:pet) ist hier eine Relation.
[4] Für ein RDF-Tripel t gilt somit: $t \in (\mathcal{IRI} \cup \mathcal{BN}) \times \mathcal{IRI} \times (\mathcal{IRI} \cup \mathcal{BN} \cup \mathcal{LIT})$.

Im Kontext der Semantic Web-Sprachen wird der Begriff *Objekt* zum einen im grammatikalischen Sinn der Subjekt-Prädikat-Objekt-Struktur der durch Tripel repräsentierten Statements verwendet. Zum anderen werden unter diesem Begriff alle Ressourcen verstanden, welche durch IRIs oder Blank Nodes repräsentiert werden. Diese Menge von Ressourcen umfasst damit i. allg. nicht nur individuelle Domänenobjekte, sondern darüberhinaus auch Elemente mehr-stelliger Relationen sowie Klassen und Relationen. Dieser zweite Objektbegriff geht in die folgende Definition ein.

Definition 2.4.

Gegeben sei ein RDF-Graph R.

Ein Tripel t mit

- $obj(t) \in LIT(R)$ heißt *Literal-Tripel*.
- $obj(t) \in IRI(R) \cup BN(R)$ heißt *Objekt-Tripel*.
- $pred(t) =$ `rdf:type` heißt *Typisierungs-Tripel*.

Für eine Ressource s in R heißt

- $R(s) := \{t \in R \,|\, subj(t) = s\}$ der *RDF-Graph von* s.[5]
- $R_{LIT}(s) := \{t \in R(s) \,|\, t$ ist Literal-Tripel$\}$ der *Literal-(RDF-)Graph von* s.
- $R_{OBJ}(s) := \{t \in R(s) \,|\, t$ ist Objekt-Tripel$\}$ heißt der *Objekt-(RDF-)Graph von* s.
- $R_{TYPE}(s) := \{t \in R(s) \,|\, t$ ist Typisierungs-Tripel$\}$ heißt der *Typisierungs-(RDF-)Graph von* s.

Für die Zusammenfassungen aller Literal-, Objekt- bzw. Typisierungs-Tripel gilt:

- $R_{LIT} := \{t \in R \,|\, t$ ist Literal-Tripel$\}$ heißt der *Literal-(RDF-)Graph von* R.
- $R_{OBJ} := \{t \in R \,|\, t$ ist Objekt-Tripel$\}$ heißt der *Objekt-(RDF-)Graph von* R.[6]
- $R_{TYPE} := \{t \in R \,|\, t$ ist Typisierungs-Tripel$\}$ heißt der *Typisierungs-(RDF)-Graph von* R.

Die Literal-Graphen sind Subgraphen, welche ausschließlich die Charakterisierung von Ressourcen durch ihre Attribute und deren Attributwerte umfassen. Die Objekt-Graphen erfassen alle sich konkret in (binärer) Relation befindlichen Ressourcenpaare. Der Objekt-Graph von R stellt topologisch gesehen das Rückrat des RDF-Graphen R dar.

Definition 2.5.

Ein *(RDF-)Vokabular* ist eine Menge von IRIs.

Eine Menge von IRIs, welche demselben (XML-)Namespace (Bray et al. 2009) zugeordnet sind, ist das *Vokabular* dieses Namespaces.[7].

Ein Vokabular, dessen Vokabularelemente die Terminologie zur Beschreibung eines domänenspezifischen Schemas bilden, nennen wir *Schema-Vokabular*.[8]

[5] entspricht dem Begriff einer sog. *entity* in Zhang et al. (2019), Def. 2.

[6] vgl. auch den Begriff des *entity-relation-graph* in Cheng et al. (2014), S. 425, welcher die relationalen Beziehungen individueller Domänenobjekte beschreibt.

[7] Wir sprechen somit z.B. vom `foaf`-Vokabular, für alles was dem `foaf`-Namespace `http://xmlns.com/foaf/0.1/` zugeordnet ist

[8] Beispiele sind die Vokabulare RDFS (Brickley and Guha 2014), RDFS-Plus (Allemang and Hendler 2011), OWL2 (Group 2012).

Listing 2.2: RDF-Graph $R^{Pets-Schema}$: Beschreibung des Schemas der Beipieldomäne "Pets" unter Verwendung des RDFS-Vokabulars (Turtle-Syntax).

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix xs: <http://www.w3.org/2001/XMLSchema#>.
@prefix gn: <http://www.geonames.org/>.
@prefix exs: <http://example.org/pets/data/1.0/> .
@prefix exd: <http://example.org/pets/schema/1.0/> .
@prefix bra: <http://www.britannica.com/animal/> .

### Classes
#
exs:Pet rdf:type rdfs:Class .
exs:Pet rdfs:isDefinedBy bra:pet .

# Class Hierarchy, is deliberately a multi-hierarchy
exs:Animal rdfs:subClassOf exs:Creature .
exs:Mammal rdfs:subClassOf exs:Animal .
exs:Pet rdfs:subClassOf exs:Animal .
exs:Cat rdfs:subClassOf exs:Mammal .
exs:Dog rdfs:subClassOf exs:Mammal .
exs:Human rdfs:subClassOf exs:Mammal .

### Properties
#
exs:age rdf:type rdf:Property .

### Properties of Properties
exs:color rdfs:range xsd:string .
exs:age   rdfs:range xsd:integer .
exs:likes rdfs:domain exs:Creature .
exs:hates rdfs:domain exs:Creature .
exs:pet rdfs:domain exs:Human .
exs:pet rdfs:range exs:Animal .
exs:livesAt rdfs:domain exs:Creature .
exs:livesAt rdfs:range exs:Address .
# Property Hierarchy
exs:loves rdfs:subPropertyOf exs:likes .
```

Üblicherweise werden in RDF-Graphen IRIs mehrerer Vokabulare verwendet. Auf diese Weise können bereits vorhandene Schemata oder Daten aus verschiedenen Domänen integriert bzw. verknüpft werden[9].

Die Namespaces können in einigen Serialisierungen durch Prefixe abgekürzt werden. Wir verwenden im Weiteren für die Namespaces bekannter Vokabulare (insbesondere die des W3C) die üblichen Prefixe, welche z.B. über die Webseite prefix.cc abgefragt werden können.

Listing 2.2 zeigt den RDF-Graphen $R^{Pets-Schema}$, welcher das zugrunde gelegte Schema für die Ressourcen des Graphen $R^{Pets-Inst}$ (siehe Listing 2.1) beschreibt. Das domänenspezifische Schema wird mittels des RDFS-Vokabulars ausgedrückt. Die domänenspezifische Terminologie fassen wir im Vokabular mit Namespace `exs` zusammen. Die Instanzen sind alle dem Namespace `exd` zugeordnet.

Mittels der Typisierungs-Tripel in $R^{Pets-Inst}_{TYPE}$ ist $R^{Pets-Inst}$ mit $R^{Pets-Schema}$ verbunden. Der RDF-

[9] siehe auch https://lod-cloud.net/

Graph $R^{Pets} = R^{Pets-Inst} \cup R^{Pets-Schema}$ umfasst unsere komplette Beispiel-Wissensbasis. Er ist nicht benamt und kann als Default-Graph betrachtet werden.

Unsere Beispiel-Wissensbasis ist bewusst unvollständig beschrieben. Zum einen fehlen zahlreiche explizite Typisierungen. Dies gilt sowohl für IRIs, welche Individuen der Domäne repräsentieren (z.B. `exs:JD`, `exd:Thing99` und `gn:2633351`), wie auch IRIs, welche Klassen, Attribute oder Relationen repräsentieren (z.B. `exs:Cat`, `exs:streetName` bzw. `exs:likes`). Zum anderen werden nicht für alle Propertys die beiden Trägermengen mittels `rdfs:domain` bzw. `rdfs:range` explizit modelliert. Diese Unvollständigkeit und Uneinheitlichkeit in den Beschreibungen reflektiert die Tatsache, dass RDF-Graphen keinem strikten Schema folgen müssen.

Im Schema-Graphen $R^{Pets-Schema}$ wurde auf die Möglichkeit der Annotationen mittels Datatype Propertys lediglich aus Gründen des Umfangs verzichtet. Grundsätzlich sind auch hier Datatype Propertys erlaubt.

Einer Ressource, exd:Kitty, wurden bewusst zwei Klassen zugeordnet, um anzudeuten, dass in RDF Multihierarchien und Mehrfachtypisierungen möglich sind.

Legt man die Semantik von RDF und RDFS zugrunde[10], können mittels Reasoning die durch unser Schema implizit ausgedrückten Typisierungen der Ressourcen unserer Beispieldomäne (Namespaces `exs:`, `exd:` und `gn:`) abgeleitet werden. Die resultierenden Tripel sind im Typisierungsgraphen $R^{Pets-Type-Inf}$ gebündelt (siehe Listing 2.3). Den Typ

`rdfs:Resource` besitzen alle IRI- und Blank Node-RDF-Terme aus R^{Pets}.

`rdfs:Class` besitzen alle RDF-Terme, welche

- in $R^{Pets-Inst}$ Objekt des Prädikats `rdf:type` oder
- in $R^{Pets-Schema}$ Subjekt oder Objekt des Prädikats `rdfs:subClassOf` oder
- in $R^{Pets-Schema}$ Objekt des Prädikats `rdfs:domain` oder `rdfs:range`

sind.

`rdf:Property` besitzen alle RDF-Terme, welche in $R^{Pets-Inst}$ Tripel-Prädikate sind.

Neben allen Individuen unserer Beispieldomäne sind auch die domänenspezifischen Klassen und Propertys in $R^{Pets-Type-Inf}$ klassifiziert (mittels `rdf:Class` bzw. `rdf:Property`). Insbesondere wird hier deutlich, dass Ressourcen i. Allg. *mehrere* Typen haben können.

Wichtig ist jedoch, dass die in Listing 2.3 gegenüber unserem RDF-Graphen R^{Pets} ergänzend vorkommenden Tripel nur dann Ausssagen über unsere kleine Beispieldomäne sind, sofern wir über RDF selbst hinaus auch die RDFS-Semantik annehmen. Dies werden wir in diesem Bericht standardmäßig nicht tun.

[10] für Hinweise bzgl. RDF- bzw. RDFS-Entailment siehe Hayes and Patel-Schneider (2014), Abschn. 8.1 und 9.2

Listing 2.3: Typisierungs-Graphen $R^{Pets-Type-Inf}$: Typisierungen, welche durch RDF- und RDFS-Reasoning über den Graphen R^{Pets} für die Ressourcen der Beipieldomäne "Pets" abgeleitet werden (Turtle-Syntax).

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix gn: <http://www.geonames.org/> .
@prefix exs: <http://example.org/pets/data/1.0/> .
@prefix exd: <http://example.org/pets/schema/1.0/> .
@prefix bra: <http://www.britannica.com/animal/> .

# Individuen
exd:Bliss rdf:type exs:Animal, exs:Creature, exs:Dog,
                                        exs:Mammal, rdfs:Resource .
exd:JD rdf:type exs:Animal, exs:Creature, exs:Human,
                                        exs:Mammal, rdfs:Resource .
exd:Kitty rdf:type exs:Animal, exs:Creature, exs:Pet, exs:Cat,
                                        exs:Mammal, rdfs:Resource .
exd:Thing99 rdf:type rdfs:Resource .
gn:2633351 rdf:type rdfs:Resource .
bra:pet rdf:type rdfs:Resource .

# Blank Nodes
_: genId1 rdf:type exs:Address, rdfs:Resource .

# Klassen
exs:Address rdf:type rdfs:Class, rdfs:Resource .
exs:Animal rdf:type rdfs:Class, rdfs:Resource .
exs:Cat rdf:type rdfs:Class, rdfs:Resource .
exs:Creature rdf:type rdfs:Class, rdfs:Resource .
exs:Dog rdf:type rdfs:Class, rdfs:Resource .
exs:Human rdf:type rdfs:Class, rdfs:Resource .
exs:Mammal rdf:type rdfs:Class, rdfs:Resource .
exs:Pet rdf:type rdfs:Class, rdfs:Resource .

# Properties
exs:age rdf:type rdf:Property, rdfs:Resource .
exs:city rdf:type rdf:Property, rdfs:Resource .
exs:color rdf:type rdf:Property, rdfs:Resource .
exs:familyName rdf:type rdf:Property, rdfs:Resource .
exs:givenName rdf:type rdf:Property, rdfs:Resource .
exs:hates rdf:type rdf:Property, rdfs:Resource .
exs:houseNumber rdf:type rdf:Property, rdfs:Resource .
exs:likes rdf:type rdf:Property, rdfs:Resource .
exs:livesAt rdf:type rdf:Property, rdfs:Resource .
exs:loves rdf:type rdf:Property, rdfs:Resource .
exs:name rdf:type rdf:Property, rdfs:Resource .
exs:pet rdf:type rdf:Property, rdfs:Resource .
exs:streetName rdf:type rdf:Property, rdfs:Resource .
gn:officialName rdf:type rdf:Property, rdfs:Resource .
gn:population rdf:type rdf:Property, rdfs:Resource .
gn:shortName rdf:type rdf:Property, rdfs:Resource .
```

3 Markierte gerichtete Multi-Graphen

In diesem Kapitel führen wir einige grundlegende Definitionen für Graphen ein, welche für die weiteren Kapitel als Basisstrukturen dienen, mittels derer RDF-Graphen und Labeled-Property-Graphen als formale Graphen aufgefasst werden können.

Alle hier betrachteten Grapharten sind gerichtete Graphen. Sie sind Multi-Graphen, d.h. das Vorkommen von mehreren gleichgerichteten Kanten zwischen einem selben Knotenpaar ist möglich (Mehrfachkanten). Schließlich können Knoten und/oder Kanten mit Markierungen versehen werden.

Grundsätzlich schränkt der Begriff *markierter Graph* die Art der verwendeten Markierung nicht ein. Im Zusammenhang mit Labeled-Property-Graphen gibt es zwei unterschiedliche Arten von Markierungen, welche beide den Knoten- bzw. Kanten zugeordnet werden können. Wir behalten hierfür die englischen Bezeichnungen bei.

Unter einem *Label* wird ein Wort (String; siehe auch Def.3.1) verstanden, mittels welchem Knoten bzw. Kanten typisiert werden können. Der zugeordnete Typ kann dabei der semantischen Klassifizierung dienen oder auch einer rein pragmatischen Bündelung von Knoten bzw. Kanten, z.B. um auf diese über einen gemeinsamen Index effizient zugreifen zu können.

Unter einer *Property* wird ein Key-Value-Paar (Schlüssel-Wert-Paar; siehe auch Def. 3.1) verstanden. Dabei bezeichnet ein *Key* eine Eigenschaft (Attribut), mittels der die durch den Knoten repräsentierte Entität bzw. die durch die Kante repräsentierte Relation näher charakterisiert wird. Ihre konkrete Ausprägung wird durch den *Value* ausgedrückt.

Die genannten Charakteristika, Gerichtetheit, Mehrfachkanten, sowie Markierung mittels Labeln und Propertys, entsprechen den von Bonifati et al. (2018), S.6, genannten. Die Autoren haben auf dieser Basis eine Definition für Property Graphen[1] vorgelegt (Abschn. 2.1, S.3). Sie zeigen auch auf, dass Spezialisierungen dieser Graphen in Gebrauch sind, welche sich durch unterschiedliche Restriktionen hinsichtlich der erlaubten Markierungen ergeben (S.6, sowie Abb.2.7, S.12). Alle diese Varianten sind jedoch gerichtete Multi-Graphen. Da wir ausschließlich solche Varianten in dieser Arbeit betrachten, verwenden wir den Begriff *Graph* fortan abkürzend für *gerichteter Multi-Graph*.

Als *First-Class-Strukturelemente* bezeichnen wir jene Strukturelemente eines Graphen, welche mit Blick auf seine Nutzung eine explizite Identität benötigen. First-Class-Citizens entsprechen damit den Strukturelementen, welche Bonifati et al. (2018) in ihren Definitionen jeweils in der Objektmenge $\mathcal{O}$ zusammenfassen. Bei der Visualisierung von Graphen sind Strukturelemente, wie bspw. Knoten, typischerweise implizit identifiziert, z.B. durch einen Kreis an einer bestimmten Position in der Darstellung. Für die textuelle Serialisierung von Graphen oder auch die informatische Behandlung von Graphen (z.B. mittels Query- oder anderen Zugriffssprachen) benötigen diese Strukturelemente einen explizit zugeordneten, eindeutigen *Identifikator* (kurz: *ID*). Die Zuordnung einer ID erfolgt mittels einer injektiven Funktion, wie von Bonifati et al. in Abschn. 2.2.3. vorgeschlagen.

In einem ersten Schritt formalisieren wir zunächst sog. *Labeled-Graphen*, welche ausschließlich mit Labeln markiert werden können. Diese erweitern wir in Abschn. 3.3 zu *Labeled-Property-Graphen*, welche ergänzend auch die Markierung mittels Propertys ermöglichen. Die *Labeled-Meta-Property-Graphen* erfassen schließlich eine Möglichkeit Propertys selbst in eingeschränkter Weise mit Propertys

[1] Sie verwenden die kürzere Bezeichnung *Property Graph* anstelle von *Labeled Property Graph*.

zu markieren.

3.1 Basisbegriffe und -notationen

In diesem Abschnitt formalisieren wir einige für diesen Bericht wichtige Grundbegriffe und Notationen.

Definition 3.1.

(b-1) Mit $\mathfrak{D}$ bezeichnen wir eine Menge von Datentypen.
Dieses können skalare Datentypen (int, ...) aber auch komplexe Datentypen (List, Set, ...) sein.

(b-2) Für jeden Datentyp $\mathcal{D} \in \mathfrak{D}$ bezeichnet $values(\mathcal{D})$ die Menge der Werte dieses Datentyps.

(b-3) $\mathcal{S}tr$ bezeichnet den Datentyp, welcher alle Strings umfasst.

(b-4) Ein *Identifikator*, kurz *ID*, ist ein Element des String-Datentyps $\mathcal{S}tr$.

(b-5) Ein *Label* ist ein Element des String-Datentyps $\mathcal{S}tr$.

(b-6) Ein *Value* ist ein Wert eines Datentyps aus $\mathfrak{D}$.

(b-7) Ein *Key* ist ein Element des String-Datentyps $\mathcal{S}tr$.
Jedem Key key ist ein Datentyp $\mathfrak{D}$ zugeordnet. Dieser wird als $dt(key)$ notiert.

(b-8) Sei $\mathcal{K}$ eine Menge von Keys und $\mathcal{V}$ eine Menge von Values.
Ein Paar $(key, val) \in \mathcal{K} \times \mathcal{V}$ heisst *Key-Value-Paar* gdw. gilt: $val \in values(dt(key))$.

Die Einbeziehung der Datentypen folgt in Anlehnung an Hartig (2014), Abschn. 4.2., Bonifati et al. (2018), Abschn. 2.2.3., sowie Chiba et al. (2020a), Def.4. Diese ist sinnvoll, weil sowohl RDF-Literale, wie auch Property-Values in realen GDB-Systemen als Ausprägungen von Datentypen beschrieben werden können.

Zu beachten ist, dass der Datentyp an den Key gebunden ist. Gemäß unserer Definition gilt für Key-Value-Paare zudem immer, dass ihr Value *typkonform* zum Datentyp des Keys ist. D.h. tritt derselbe Key key mehrmals in einem oder verschiedenen Knoten auf (siehe später LP-Graphen, Abschn. 3.3), haben die jeweils assoziierten Values immer den Typ $dt(key)$. Diese Bedingung stellt nicht zwingend eine Einschränkung dar, weil in vielen Typsystemen ein umfassender Basistyp vorhanden ist, aus dem alle anderen Typen abgeleitet sind (analog zu `Object` in Java oder `rdfs:Resource` in RDF).

Zur Visualisierung von Graphen verwenden wir in diesem Bericht die in Abb.3.1 verwendete Bildsprache. Identifikatoren von Knoten bzw. Kanten zeigen wir nur dann in der Visualisierung an, wenn zur eindeutigen Referenzierung erforderlich oder spezifische Identifikatoren zugewiesen wurden.

3.2 Labeled-Graphen

Graphen, bei denen ausschließlich Label zur Markierung verwendet werden können, sind Gegenstand dieses Abschnitts.

Wir beginnen mit Graphen, deren Markierungen auf genau ein Label pro Kante beschränkt sind, Knoten sind hier unmarkiert. Zwischen einem Knotenpaar können mehrere Kanten existieren, sofern deren Label verschieden sind.

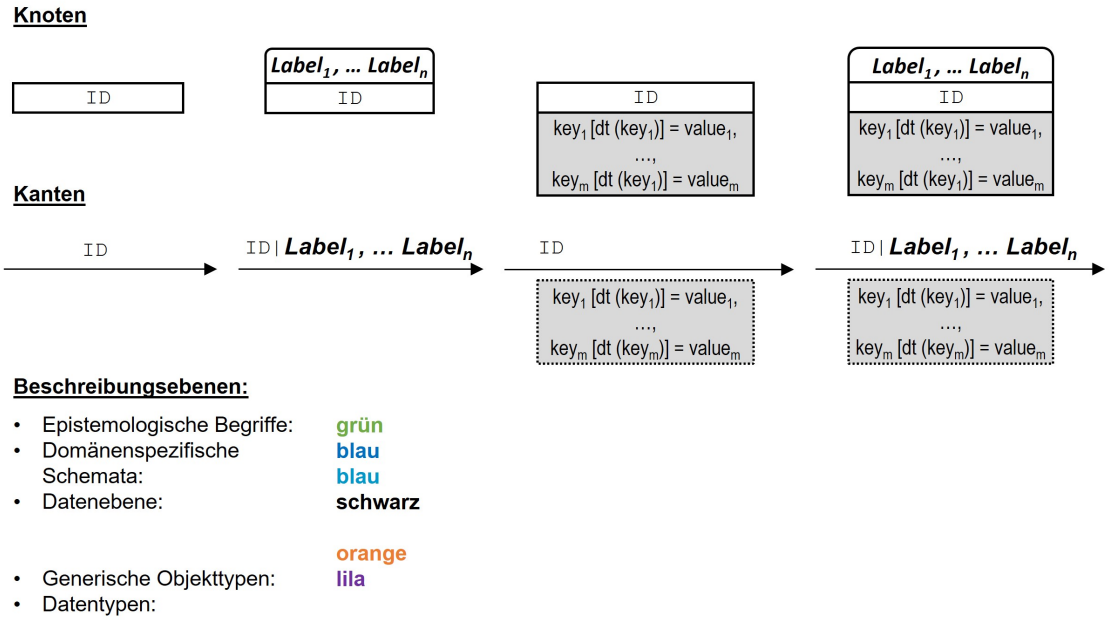

Abb. 3.1: Bildsprache für die Visualisierung formaler Graphen: a) die Darstellung von Knoten und Kanten mit Identifikatoren und/oder Labeln und/oder Key-Value-Paaren und b) die farbliche Codierung von Begriffen bzw. visuellen Elementen gemäß Zugehörigkeit der vorhandenen Beschreibungsebenen.

Eine gebräuchliche einfache Formalisierung für solche kantenmarkierten Multi-Graphen konstituiert die Kanten indirekt über die Knoten. Um diesen Aspekt hervorzuheben, bezeichnen wir solche Graphen in diesem Bericht als *Node-Pair-Labeled-Graph*, anstatt wie sonst üblich als *edge-labeled*.

Definition 3.2.

Ein *Node-Pair-Labeled-Graph*, kurz *NPL-Graph* ist ein gerichteter (kanten-) markierter Multi-Graph

$$NPLG = (V, E, \mathcal{L}_E, id_V),$$

für den gilt:

(m-1) V ist eine endl. Menge von *Knoten*.
(Sofern zur textuellen oder visuellen Beschreibung der Knoten notwendig, verwenden wir für sie die Bezeichner $v_1, v_2, v_3, ...$)

(m-2) $\mathcal{L}_E$ ist eine Menge von Label zur Markierung der Kanten.

(m-3) $E \subseteq (V \times \mathcal{L}_E \times V)$ ist eine endl. Menge von *Kanten*;
Für eine Kante $(v_1, l, v_2) \in E$ bezeichnet $src(e) = v_1$ den *Start-* oder *Quellknoten* der Kante e und $trg(e) = v_2$ deren *End-* oder *Zielknoten*.

(f-1) id_V ist eine (partielle) injektive Funktion, welche jedem Knoten eine eindeutige ID zuweist.
(Mittels dieser Funktion können spezifische Identifikatoren zugewiesen werden)

Knoten sind in dieser Modellierung die einzigen *First-Class-Strukturelemente* des Graphen. Nur ihnen können eigene Bezeichner (*Identifikatoren*) zugewiesen werden. Die Kanten werden dagegen durch Knotenpaare und ein Label identifiziert.

Abb. 3.2 zeigt einen Beispiel-NPL-Graphen gemäß obiger Definition ohne spezifisch zugewiesene Identifikatoren.

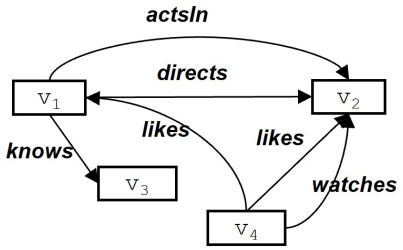

Abb. 3.2: Visualisierung eines Node-Pair-Labeled-Graphen (vgl. Def.3.2)

Mit der folgenden Definition erweitern wir nun die möglichen Markierungen um Label für Knoten. Zudem wird die Beschränkung auf genau ein Label als Markierung aufgehoben. Knoten *und* Kanten sind eigenständig identifizierbare Strukturelemente, also beides First-Class-Strukturobjekte. Dies erlaubt uns später eine einfachere Erweiterung um weitere Markierungen.

Mit dieser Definition ist eine einheitliche umfassende Strukturbeschreibung gerichteter Multi-Graphen gegeben, welche mittels Labeln markiert werden können.

Die Definition erlaubt es uns,

- unterschiedliche in der Literatur beschriebene Formalisierungen von RDF-Graphen als Labeled-Graph zu formulieren (siehe Kap. 6), und
- konsistent auf Labeled-Property-Graphen (siehe Abschn. 3.3 und deren relevante Varianten zu erweitern.

Wir führen ergänzend eine Nomenklatur ein, um verschiedene Varianten solcher Graphen typisieren zu können.

Definition 3.3.

Ein *Labeled-Graph*, kurz *L-Graph*, ist ein gerichteter (knoten- und kanten-) markierter (Multi-)Graph

$$LG = (V, E, \mathcal{L}_V, \mathcal{L}_E, id_V, id_E, vrt, lbl_V, lbl_E)$$

für den gilt:

(m-1) V ist eine endl. Menge von sog. *Knoten*.
(Sofern zur textuellen oder visuellen Beschreibung der Knoten notwendig, verwenden wir für sie die Bezeichner $v_1, v_2, v_3, ...$)

(m-2) E ist eine endl. Menge von sog. *Kanten*.
(Sofern zur textuellen oder visuellen Beschreibung der Kanten notwendig, verwenden wir für sie die Bezeichner $e_1, e_2, e_3, ...$)

(nb-3) V und E sind disjunkt, d.h. $V \cap E = \emptyset$;

(m-3) $\mathcal{L}_V$ ist eine Menge von *Knoten-Labeln*.

(m-4) $\mathcal{L}_E$ ist eine Menge von *Kanten-Labeln*.

(f-1) id_V ist eine (partielle) injektive Funktion, welche Knoten eine eindeutige ID zuweist. Wird id_V nicht explizit angegeben, so gilt defaultmäßig $dom(idV) = \emptyset$.
(Mittels dieser Funktion können spezifische Identifikatoren zugewiesen werden. Im Defaultfall werden die Bezeichner $v_1, v_2, ...$ (s.o.) verwendet.)

(f-2) id_E ist eine (partielle) injektive Funktion, welche Kanten eine eindeutige ID zuweist. Wird id_E nicht explizit angegeben, so gilt defaultmäßig $dom(idE) = \emptyset$.
(Mittels dieser Funktion können spezifische Identifikatoren zugewiesen werden. Im Defaultfall werden die Bezeichner $e_1, e_2, ...$ (s.o.) verwendet.)

(f-3) $vrt : E \to V \times V$ ist eine totale Funktion, die jeder Kante eine geordnetes Paar (v_1, v_2) von Knoten zuordnet.
Für eine Kante $e \in E$ mit $vrt(e) = (v_1, v_2)$ bezeichnet $src(e) := v_1$ den *Start-* oder *Quellknoten* der Kante e und $trg(e) := v_2$ deren *End-* oder *Zielknoten*.

(f-4) $lbl_V : V \to \mathfrak{P}_{endl}(\mathcal{L}_V)$ ist eine Funktion, welche jedem Knoten genau eine endl. Menge von (Knoten-)Labeln zuordnet.

(f-5) $lbl_E : E \to \mathfrak{P}_{endl}(\mathcal{L}_E)$ ist eine Funktion, welche jeder Kante genau eine endl. Menge von (Kanten-)Labeln zuordnet.

Soweit nicht explizit anders festgelegt, werden wir für Knoten von L-Graphen als IDs jeweils Strings der Form v_1, v_2, v_3 usw. sowie für Kanten die Bezeichnungen e_1, e_2, e_3 usw. verwenden.

Abb. 3.2 zeigt einen Beispielgraphen, bei dem kein bis mehrere Label an den Knoten und mindestens ein Label an den Kanten auftreten.

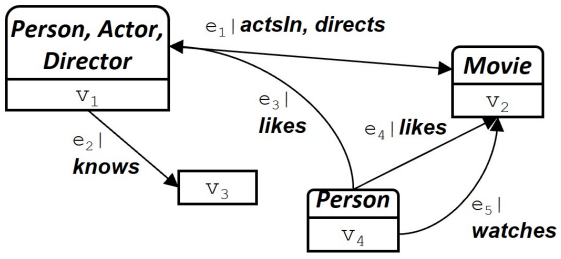

Abb. 3.3: Visualisierung eines Labeled-Graphen vom Typ $(*, +)$ (vgl. Def. 3.3 und 3.4)

Zur Charakterisierung verschiedener Varianten von Labeled-Graphen führen wir eine Typ-Notation ein, welche die Anzahl der Label, welche Knoten oder Kanten zugeordnet werden können, differenziert berücksichtigt.

Definition 3.4.

Sei $LG = (V, E, \mathcal{L}_V, \mathcal{L}_E, id_V, id_E, vrt, lbl_V, lbl_E)$ ein Labeled-Graph.
Seien $\#vl, \#el \in \{0, 1, +, *\}$. Für $\#vl$ gelte:

$\#vl = 0$ *gdw.* $\forall v \in V : |lbl_V(v)| = 0$
(Jedem Knoten ist kein Label *zugeordnet, d.h. also alle Knoten sind unmarkiert.)*

$\#vl = 1$ *gdw.* $\forall v \in V : |lbl_V(v)| = 1$
(Jedem Knoten ist genau ein Label *zugeordnet.)*

$\#vl = \leq 1$ *gdw.* $\forall v \in V : |lbl_V(v)| \leq 1$
(Jedem Knoten ist maximal ein Label *zugeordnet.)*

$\#vl = +$ *gdw.* $\forall v \in V : |lbl_V(v)| \geq 1$
(Jedem Knoten ist mindestens ein Label *zugeordnet.)*

$\#vl = *$ *gdw.* $\forall v \in V : |lbl_V(v)| \geq 0$
(Jedem Knoten können beliebig viele Labels *zugeordnet werden.)*

Für $\#el$ gelte die analoge Festlegung mit Bezug zu lbl_E.

Wir nennen $\#vl$ und $\#el$ *Knoten-* bzw. *Kanten-Label-Typ* von LG und sagen, LG hat den *(Label-)Typ* $(\#vl, \#el)$.

Für Labeled-Graphen LG, deren Typ $(\#vl, \#el)$ bekannt ist, schreiben wir auch $LG_{(\#vl,\#el)}$, um

ihren Typ direkt sichtbar zu machen.

Gemäß dieser Festlegung können jetzt Varianten von Labeled-Graphen charakterisiert werden. Hier einige Beispiele:

$LG_{(1,0)}$: bezeichnet einen ausschließlich knotenmarkierten Graphen LG, bei dem alle Knoten genau ein Label besitzen.

$LG_{(0,1)}$: bezeichnet einen ausschließlich kantenmarkierten Graphen LG, bei dem alle Kanten genau ein Label besitzen.

$LG_{(*,*)}$: bezeichnet einen knoten- und kantenmarkierten Graphen LG, bei dem Knoten und Kanten beliebig viele Label zugeordnet werden können.

$LG_{(+,0)}$: bezeichnet einen ausschließlich knotenmarkierten Graphen LG, bei dem jeder Knoten mindestens ein Label besitzt.

3.3 Labeled-Property-Graphen

Labeled-Property-Graphen sind Erweiterungen von Labeled-Graphen, welche ergänzend die Markierung von Knoten und Kanten mit Propertys erlauben. Es handelt sich damit weiterhin um gerichtete knoten- und kantenmarkierte Graphen.

Da es für das Labeled-Property-Graph-Modell keine Standardisierung gibt, fallen informelle Beschreibungen wie auch Formalisierungen im Detail inhaltlich unterschiedlich aus (vgl. z.B. Hartig (2014), Robinson et al. (2015), Rodriguez (2015), Bonifati et al. (2018), Kap. 2, Besta et al. (2020), Angles et al. (2020b), Angles et al. (2020a), Thakkar (2021)).

Auf der Website des *Graph Computing Frameworks Apache TinkerPop™*[2] werden (Labeled-)Property-Graphen z.B. wie folgt beschrieben:

> *"A graph is a structure composed of vertices and edges. Both vertices and edges can have an arbitrary number of key/value-pairs called properties. Vertices denote discrete objects such as a person, a place, or an event. Edges denote relationships between vertices. For instance, a person may know another person, have been involved in an event, and/or have recently been at a particular place. Properties express non-relational information about the vertices and edges. Example properties include a vertex having a name and an age, and an edge having a timestamp and/or a weight. Together, the aforementioned graph is known as a property graph and it is the foundational data structure of Apache TinkerPop™."*

Label werden hier nicht erwähnt, obwohl sie durch das Framework unterstützt werden.

In der Dokumentation der Multi-Modell-Datenbank OrientDB[3] heißt es[4]: *"OrientDB's graph model is represented by the concept of a property graph"*. Neben der Zuordnung von beliebig vielen Propertys an Knoten und Kanten, sind nur Kanten mit genau einem Label versehen.

Gemäß Hartig (2014) ist im Buch *Graph Databases* von Robinson et al. (2013) eine der repräsentativsten informellen Beschreibungen zu finden. Wir zitieren hier die Beschreibung aus der neueren Ausgabe (Robinson et al. (2015), S.4):

> *"A labeled property graph has the following characteristics:*
>
> - *It contains nodes and relationships.*

[2] http://tinkerpop.apache.org/; Zugriff 4.11.2021
[3] https://orientdb.org
[4] https://orientdb.org/docs/3.0.x/datamodeling/Tutorial-Document-and-graph-model.html

- *Nodes contain properties (key-value-pairs).*
- *Nodes can be labeled with one or more labels.*
- *Relationships are named and directed, and always have a start and end node.*
- *Relationships can also contain properties.*

"

Die nun folgende Formalisierung von Labeled-Property-Graphen ist durch die oben genannten Arbeiten inspiriert.

Sie soll

- zum einen die verschiedenen in diesen Arbeiten vorgestellten Graphmodelle so subsumieren, dass diese als Spezialfälle dieser Definition eingeordnet werden können und
- zudem eine nahtlose Erweiterung unserer Definition von Labeled-Graphen sein (Def. 3.3).

Definition 3.5.

Ein *Labeled-Property-Graph*, kurz *LP-Graph*, ist ein gerichteter (knoten- und kanten-) markierter (Multi-)Graph

$$LPG = (V, E, \mathcal{L}_V, \mathcal{L}_E, id_V, id_E, vrt, lbl_V, lbl_E, \mathcal{K}, \mathcal{V}, P, kv_P, prp_V, prp_E)$$

für den gilt:

(g-1) $(V, E, \mathcal{L}_V, \mathcal{L}_E, id_V, id_E, vrt, lbl_V, lbl_E)$ ist ein Labeled-Graph.

(m-5) $\mathcal{K}$ ist eine endl. Menge von *Keys*.

(m-6) $\mathcal{V}$ ist eine endl. Menge von *Values*.

(m-7) P ist eine endl. Menge von *Propertys*.
(Sofern zur textuellen oder visuellen Beschreibung der Propertys notwendig, verwenden wir für sie die Bezeichner $p_1, p_2, p_3, ...$)

(f-6) $kv_P : P \to \mathfrak{P}_{endl}(\mathcal{K} \times \mathcal{V})$ ist eine Funktion, welche jeder Property $p \in P$ genau ein (typkorrektes) Key-Value-Paar (key_p, val_p) zuordnet.
(Verschiedene Propertys können ein gleiches Key-Value-Paar zugeordnet bekommen.)

(f-7) $prp_V : V \to \mathfrak{P}_{endl}(P)$ ist eine Funktion für die gilt:
$prp_V(v_1) \cap prp_V(v_2) = \emptyset$, für alle $v_1, v_2 \in V$ mit $v_1 \neq v_2$.
(Jedem Knoten wird genau eine endl. Menge von Propertys zuordnet. Verschiedene Knoten haben keine gemeinsamen Propertys.)

(f-8) $prp_E : E \to \mathfrak{P}_{endl}(P)$ ist eine Funktion, für die gilt:
$prp_E(e_1) \cap prp_E(e_2) = \emptyset$, für alle $e_1, e_2 \in E$ mit $e_1 \neq e_2$.
(Jeder Kante wird genau eine endl. Menge von Propertys zuordnet. Verschiedene Kanten haben keine gemeinsamen Propertys.)

(m-8) $props(V) := \{p \in P \mid \exists v \in V : prp_V(v) = p\}$ ist die Menge aller den Knoten zugeordneten Propertys.

(m-9) $props(E) := \{p \in P \mid \exists e \in E : prp_E(e) = p\}$ die Menge aller den Kanten zugeordneten Propertys.

(nb-2) $props(V) \cap props(E) = \emptyset$.
(Eine Property kann nicht Kanten- und Knoten-Property zugleich sein.)

Die Anforderungen (f-7), (f-8) und (nb-2) stellen sicher, dass eine Property entweder nur genau einem Knoten oder genau einer Kante zugeordnet sein kann. Wichtig ist jedoch: Diese Einschränkung

erlaubt, dass ein und dasselbe Key-Value-Paar Propertys verschiedener Knoten, verschiedener Kanten oder sogar auch von Knoten und Kanten zugeordnet werden kann.

Um auch diesen Graph-Typ leichter erweitern zu können, sind Propertys nicht einfach als Key-Value-Paare, sondern als eigenständige First-Class-Strukturobjekte modelliert, denen genau ein Key-Value-Paar zugeordnet wird.

Für die Propertys legen wir fest:

Definition 3.6.

Sei p eine Property, der das Key-Value-Paar $(key_p, value_p)$ zugeordnet wurde.

$key(p) := key_p$ bezeichnet den *Property-Key* und $val(p) := value_p$ den *Property-Value* von p.

Sei P eine Menge von Propertys. $keys(P) = \{key(p) \,|\, p \in P\}$ bezeichnet die Menge der (verschiedenen) mit P verknüpften Property-Keys.

Wir führen für Labeled-Property-Graphen ebenfalls Typen ein, um Varianten besser unterscheiden zu können:

Definition 3.7.

Sei $LPG = (V, E, \mathcal{L}_V, \mathcal{L}_E, id_V, id_E, vrt, lbl_V, lbl_E, \mathcal{K}_V, \mathcal{K}_E, \mathcal{V}, prp_V, prp_E)$ ein Labeled-Property-Graph.
Seien $\#vp, \#ep \in \{0, 1, +, *, +u, *u\}$. Für $\#vp$ gelte:

$\#vp = 0$ *gdw.* $\forall v \in V : |prp_V(v)| = 0$
(Jedem Knoten ist keine Property *zugeordnet.)*

$\#vp = 1$ *gdw.* $\forall v \in V : |prp_V(v)| = 1$
(Jedem Knoten ist genau eine Property *zugeordnet.)*

$\#vp = +$ *gdw.* $\forall v \in V : |prp_V(v)| \geq 1$
(Jedem Knoten ist mindestens eine Property *zugeordnet.)*

$\#vp = *$ *gdw.* $\forall v \in V : |prp_V(v)| \geq 0$
(Jedem Knoten können beliebig viele Propertys *zugeordnet werden.)*

$\#vp = +u$ *gdw.* $\forall v \in V : (|prp_V(v)| \geq 1 \wedge |keys(prp_V(v))| = |prp_V(v)|)$
(Jedem Knoten ist mindestens eine Property *zugeordnet und alle diese Propertys besitzen unterschiedliche Keys.)*

$\#vp = *u$ *gdw.* $\forall v \in V : (|prp_V(v)| \geq 0 \wedge |keys(prp_V(v))| = |prp_V(v)|)$,
(Jedem Knoten können beliebig viele Propertys *zugeordnet werden, welche alle unterschiedliche Keys besitzen müssen.)*

Für $\#ep$ gelte die analoge Festlegung mit Bezug zu prp_E.

Wir nennen $\#vp$ und $\#ep$ den *Knoten-* bzw. *Kanten-Property-Typ* von LPG und sagen LPG hat den *Property-Typ* $(\#vp, \#ep)$.

Sei $(\#vl, \#el)$ der Label-Typ des in LPG gemäß Def. 3.5, (g-1) eingebetteten Labeled-Graphen. Dann hat auch LPG den *Label-Typ* $(\#vl, \#el)$.

Insgesamt ist LPG vom *Typ* $\frac{(\#vl,\#el)}{(\#vp,\#ep)}$.

Für Labeled-Property-Graphen LPG, deren Typ $\frac{(\#vl,\#el)}{(\#vp,\#ep)}$ bekannt ist, schreiben wir auch $LPG_{\frac{(\#vl,\#el)}{(\#vp,\#ep)}}$, um ihren Typ direkt sichtbar zu machen.

Betrachten wir wieder einige Typ-Beispiele für Labeled-Property-Graphen:

$LPG_{\frac{(1,1)}{(*,*)}}$: bezeichnet einen Graphen, bei dem alle Knoten und Kanten genau ein Label besitzen. Jedem/r Knoten bzw. Kante können beliebig viele Propertys zugeordnet werden.

$LPG_{\frac{(+,1)}{(*,*)}}$: bezeichnet einen Graphen, bei dem alle Knoten mindestens ein Label besitzen müssen, Kanten jedoch nur genau eins. Jedem/r Knoten bzw. Kante können beliebig viele Propertys zugeordnet werden.

$LPG_{\frac{(1,1)}{(*u,0)}}$: bezeichnet einen Graphen, bei dem alle Knoten und Kanten genau ein Label besitzen. Jedem Knoten können beliebig viele Propertys zugeordnet werden, welche jedoch alle verschiedene Keys besitzen müssen. Kanten haben keine Propertys.

Unsere Formalisierung trägt der Vorstellung Rechnung, dass Labeled-Property-Graphen Erweiterungen von Labeled-Graphen um einen weiteren Markierungstyp sind. Def.3.5 und 3.7 sind in diesem Sinne nahtlose Erweiterungen der Def. 3.3 und 3.4. Ein Labeled-Property-Graph
$LPG = (V, E, \mathcal{L}_V, \mathcal{L}_E, id_V, id_E, vrt, lbl_V, lbl_E, \mathcal{K}_V, \mathcal{K}_E, \mathcal{V}, prp_V, prp_E)$ mit Typ $\frac{(\#vl,\#el)}{(0,0)}$
entspricht dem in ihm eingebetteten Labeled-Graphen
$(V, E, \mathcal{L}_V, \mathcal{L}_E, id_V, id_E, vrt, lbl_V, lbl_E)$ mit Typ $(\#vl, \#el)$,
da weder Knoten noch Kanten in LPG Propertys besitzen.

Definition 3.8.

Ein *Labeled-Meta-Property-Graph*, kurz *LMP-Graph*, ist ein gerichteter (knoten- und kanten-) markierter (Multi-)Graph
$LMPG = (V, E, \mathcal{L}_V, \mathcal{L}_E, id_V, id_E, vrt, lbl_V, lbl_E, \mathcal{K}, \mathcal{V}, P, kv_P, prp_V, prp_E, prp_P)$
für den gilt:

(g-1) $(V, E, \mathcal{L}_V, \mathcal{L}_E, id_V, id_E, vrt, lbl_V, lbl_E, \mathcal{K}, \mathcal{V}, P, kv_P, prp_V, prp_E)$ ist ein Labeled-Property-Graph.

(f-9) $prp_P : (props(V) \cup props(E)) \to \mathfrak{P}_{endl}((P \backslash (props(V) \cup props(E))))$ ist eine Funktion für die gilt:
$prp_P(p_1) \cap prp_P(p_2) = \emptyset$, für alle $p_1, p_2 \in (props(V) \cup props(E))$ mit $p_1 \neq p_2$.
(Jeder Knoten- oder Kantenproperty wird genau eine endl. Menge von Meta-Propertys zugeordnet. Verschiedene Propertys haben keine gemeinsamen Meta-Propertys.)

(m-12) $props(P) := \{m \in P \mid \exists p \in (props(V) \cup props(E)) : prp_P(m) = p\}$ ist die Menge aller den Knoten- und Kanten zugeordneten Meta-Propertys.
(Die Menge der Meta-Propertys ist disjunkt zu den Kanten- und Knotenproperties. Meta-Properties können keine Propertys zugeordnet werden.)

Wir beziehen wie folgt die Meta-Propertys in unser Typisierungsschema ein:

Definition 3.9.

Sei $LMPG = (V, E, \mathcal{L}_V, \mathcal{L}_E, id_V, id_E, vrt, lbl_V, lbl_E, \mathcal{K}, \mathcal{V}, P, kv_P, prp_V, prp_E, prp_P)$ ein Labeled-Meta-Property-Graph.
Seien $\#vmp, \#emp \in \{M0, M1, M+, M*, M+u, M*u\}$. Für $\#vmp$ gelte (analog zu Def.3.7):

$\#vmp = M0$ *gdw.* $\forall p \in (props(V)) : |prp_P(p)| = 0$
(Jeder Knoten-Property ist keine Meta-Property *zugeordnet.)*

$\#vmp = M1$ *gdw.* $\forall p \in (props(V)) : |prp_P(p)| = 1$
(Jeder Knoten-Property ist genau eine Meta-Property *zugeordnet.)*

$\#vmp = M+$ *gdw.* $\forall p \in (props(V)) : |prp_P(p)| \geq 1$
(Jeder Knoten-Property ist mindestens eine Meta-Property *zugeordnet.)*

$\#vmp = M*$ *gdw.* $\forall p \in (props(V)) : |prp_P(p)| \geq 0$
(Jeder Knoten-Property können beliebig viele Meta-Propertys *zugeordnet werden.)*

$\#vmp = M + u$ *gdw.* $\forall p \in P : (|prp_P(v)| \geq 1 \quad \cap |keys(prp_P(p))| = |prp_P(p)|)$
(Jeder Knoten-Property ist mindestens eine Meta-Property *zugeordnet und alle diese Propertys besitzen unterschiedliche Keys.)*

$\#vmp = M * u$ *gdw.* $\forall p \in P : (|prp_P(p)| \geq 0 \quad \cap |keys(prp_P(p))| = |prp_P(p)|)$,
(Jeder Knoten-Property können beliebig viele Meta-Propertys *zugeordnet werden, welche alle unterschiedliche Keys besitzen müssen.)*

Für $\#emp$ gelte die analoge Festlegung mit Bezug zu $props(E)$.

Wir nennen $\#vmp$ und $\#emp$ *Knoten-* bzw. *Kanten-Meta-Property-Typ* von $LMPG$ und sagen $LMPG$ hat den *Meta-Property-Typ* $(\#vmp, \#emp)$.

Sei $\frac{(\#vl,\#el)}{(\#vp,\#ep)}$ der Typ des gemäß Def. 3.8 (g-1) in $LPMG$ eingebetten Labeled-Property-Graphen. $LMPG$ ist dann vom Typ *Typ* $\frac{(\#vl,\#el)}{(\#vp|\#vmp,\#ep|\#emp)}$.

Für Labeled-Meta-Property-Graphen LPG, deren Typ $\frac{(\#vl,\#el)}{(\#vp|\#vmp,\#ep|\#emp)}$ bekannt ist, schreiben wir auch $LPMG_{\frac{(\#vl,\#el)}{(\#vp|\#vmp,\#ep|\#emp)}}$, um ihren Typ direkt sichtbar zu machen.

3.4 Labeled-Property-Graphen - Vielfalt in Literatur und GDB-Systemen

Wir hatten eingangs darauf hingewiesen, dass unsere Formalisierung durch zahlreiche Arbeiten in der Literatur inspiriert ist. Wir möchten hier aufzeigen, wie sich dort gefundene Graph-Definitionen mittels unserer Typ-Notation kennzeichnen und ordnen lassen.

Besta et al. (2020) verstehen unter dem *Labeled Property Graph Model* einen LP-Graphen des Typs $\frac{(*,*)}{(*,*)}$. In diesem Sinn interpretieren auch Angles (2018) ihre in Definition 2. spezifizierten *property graphs* und Chiba et al. (2020a) den Begriff *Property Graph Model* (Definition 4). Chiba et al. beziehen ergänzend auch ungerichtete Kanten als eigene Kantenobjekte ein. Wir betrachten ihr Modell jedch als kompatibel mit unserer LP-Graph-Definition, da ungerichtete Kanten durch zwei gegengerichtete identisch markierte Kanten darstellbar sind.

Übersetzt man die informelle Beschreibung aus dem Buch von Robinson et al. (2015), so ergibt sich der LP-Graph-Typ $\frac{(*,1)}{(*,*)}$.

Unsere Bezeichnungen $+u$ bzw. $*u$ zur Charakterisierung der Property-Typen folgen Hartig (2014). Er nennt solche Graphen *property-unique* (Def.3). Die von ihm beschriebenen Property Graphen sind gemäß unserer Formalisierung LP-Graphen vom Typ $\frac{(0,1)}{(*,*)}$ und, sofern sie property-unique sind, vom Typ $\frac{(0,1)}{(*u,*u)}$.

Bonifati et al. (2018) widmen sich ausführlich der formalen Beschreibung von Property Graph-Modellen und ordnen diese in einer Hierarchie. Die Definition ihres *property graphs* beschreibt einen LP-Graphen des Typs $\frac{(*,*)}{(*u,*u)}$. Sie formalisieren darüberhinaus interessante Erweiterungen, welche Pfade, Subgraphen, sowie Hyperknoten und -kanten als markierbare Strukturobjekte einbeziehen.

Bonifati et al. (2018) ordnen auch das von Neo4J[5], Oracle/PGQL[6] sowie das von Apache TinkerPop™ unterstützte Graph-Modell als Spezialisierungen ihres *Property Graph Models* ein. Danach

[5] https://neo4j.com/
[6] siehe https://pgql-lang.org/ sowie https://docs.oracle.com/en/database/oracle/property-graph/20.4/spgdg/property-graph-query-language-pgql.html

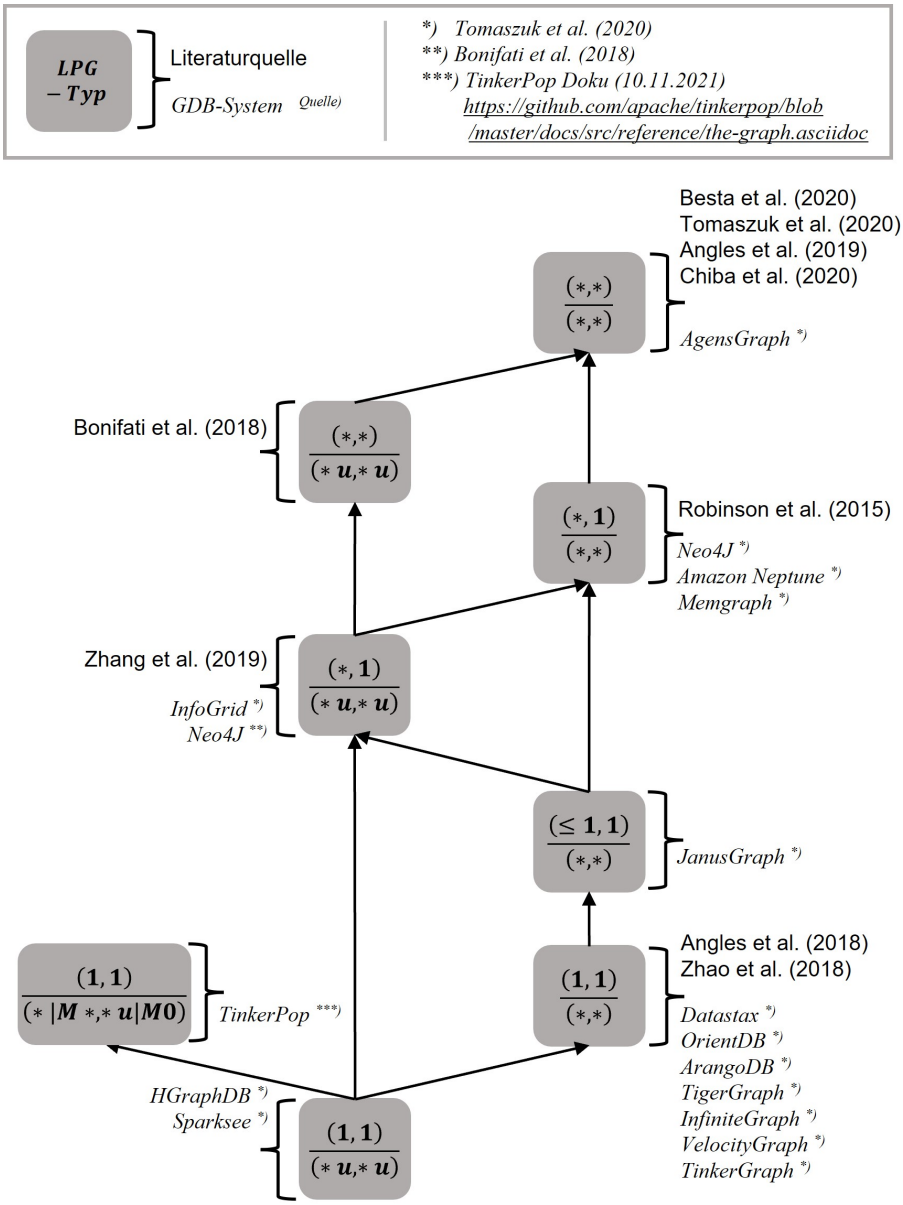

Abb. 3.4: Hierarchie von (M)LP-Graph-Typen, welche in der Literatur bzw. von GDB-Systemen unter dem einheitlichen Begriff *(Labeled-)Property-Graph* gefasst werden.

würden Neo4J und Oracle/PGQL unserem Typ $\frac{(*,1)}{(*u,*u)}$ und TinkerPop dem Typ $\frac{(1,1)}{(*u,*u)}$ entsprechen. Bzgl. dieser Einordnung gibt es inzwischen Abweichungen zur neueren Aufbereitung von Tomaszuk et al. (2020), wonach in Neo4J-Graphen keine *unique properties* zwingend sind. In TinkerPop™-Graphen müssen nach unserer Recherche[7] nur die Kanten-Properties unique sein. Zudem ermöglicht Apache TinkerPop™die Markierung von Knoten-Properties mit sog. *meta properties*, was uns zur erweiterten Definition Def. 3.8 und entsprechender Einordnung der TinkerPop-Graphen mit dem Typ $\frac{(1,1)}{(*|M*,*u|M0)}$ motiviert hat.

Zhang et al. (2019) verstehen unter einem *Property Graph* einen LP-Graphen des Typs $\frac{(*,1)}{(*u,*u)}$.

Zhao et al. (2018) beschränken dagegen auch die Anzahl der Label an Knoten und definieren *Labeled Property Graphs* als LP-Graphen des Typs $\frac{(1,1)}{(*,*)}$.

Auch die von Angles et al. (2020b) bzw. Thakkar (2021) spezifizierten *Property Graphs* (siehe Thakkar Abschn. C, Def. 8, bzw. Kap.4, Def. 6) sind LP-Graphen des Typs $\frac{(1,1)}{(*,*)}$. Die Autoren formalisieren neben Knoten und Kanten ergänzend auch Propertys als First-Class-Strukturobjekte.

Tomaszuk et al. (2020) haben in ihrem Artikel 15 Graphdatenbanken hinsichtlich des von ihnen unterstützten Property-Graph-Modells analysiert und in Tab. 1 für jedes System die verwendete Variante erfasst. Es fällt auf, dass bei all diesen Systemen kein Unterschied zwischen Knoten und Kanten hinsichtlich der Anzahl zuweisbarer Propertys gemacht wird. Bzgl. der Labelzuordnung bestehen solche Unterschiede durchaus. Um alle Varianten der untersuchten Systeme zu erfassen, definieren die Autoren *Property Graphs* als LP-Graphen des Typs $\frac{(*,*)}{(*,*)}$. Das entpricht damit der Sichtweise von Besta et al. (2020), Angles (2018) und Chiba et al. (2020a) (s.o.) und unserem allgemeinsten LP-Graph-Typ.

In Abb. 3.4 haben wir in der Literatur beschriebene Labeled-Property-Graph-Definitionen und -Systeme auf Basis unserer Typ-Notation eingeordnet. Dabei sind die Typen der LP-Graph-Systeme weitestgehend Tomaszuk et al. (2020) entnommen.

[7] gem. https://github.com/apache/tinkerpop/blob/master/docs/src/reference/the-graph.asciidoc; Zugriff 10.11.2021

4 Der Graphbegriff in den RDF-Recommendations des W3C

Wie in Kap. 2 beschrieben sind RDF-Graphen Tripelmengen. Eine Formalisierung i.S. der Graphentheorie erfolgt in den RDF Recommendations des W3C interessanter Weise nicht. Es gibt lediglich einige Hinweise, wie die Tripel als Konstituenten eines formalen Graphen verstanden werden sollen.

Im *RDF Primer 1.1* (Schreiber and Raimond 2014), dem zentralen Einführungsdokument zu RDF, heisst es in Abschn. 3.1 im Zusammenhang mit der Beschreibung des RDF-Datenmodells:

> *"We can visualize triples as a connected graph. Graphs consists of nodes and arcs. The subjects and objects of the triples make up the nodes in the graph; the predicates form the arcs."*

In der Recommendation *RDF 1.1 Concepts and Abstract Syntax* (Cyganiak et al. 2014) lautet es in Abschn. 1.1 *Graph-based Data Model*:

> *"An RDF graph can be visualized as a node and directed-arc diagram, in which each triple is represented as a node-arc-node link. [...] There can be three kinds of nodes in an RDF graph: IRIs, literals, and blank nodes."*

und später in Abschn. 3.1:

> *"The set of nodes of an RDF graph is the set of subjects and objects of triples in the graph."*

Das Konzept eines Graphen wird also eher als eine Visualisierungsmöglichkeit eingeführt denn als formale Struktur.

Bereits 2004 stellten dies Hayes and Gutiérrez (S.1) hinsichtlich der ersten RDF Recommendation (Klyne and Carroll 2004) fest:

> *"Currently, the RDF specification documents do not distinguish clearly among the term "RDF Graph", the mathematical concept of graph, and the graph-like visualization of RDF data."*

In seiner Diplomarbeit zitiert Hayes (2004, S.50) ebenfalls obige Charakterisierung der Knotenmenge als die Menge der Subjekte und Objekte von Tripeln im RDF-Graphen. Sie ist so bereits in der ersten Recommendation von Klyne and Carroll (2004) vorhanden. Er kommentiert:

> *"which is somewhat fragmentary, as no definition for the edges is given."*

Aus den genannten Hinweisen der W3C Recommendations ergeben sich folgende Erkenntnisse für die Beschreibung eines RDF-Graphen R als formaler Graph $G(R)$:

(1) Die Mengen $IRI(R)$, $BN(R)$ und $LIT(R)$ werden als disjunkt angesehen (Cyganiak et al. 2014, Abschn. 3.1)

(2) Jede IRI, jeder Blank Node und jedes Literal, welche/r/s als Tripel-Subjekt oder Tripel-Objekt in R vorkommt, ist ein Knoten des Graphen $G(R)$. Die IRIs, lokalen Blank Node-Bezeichner bzw. Literale *identifizieren* diese Knoten und sind nicht deren Markierung.

(s.o. *" The set of nodes [...]* **is** *the set of subjects and objects of triples [...]. "* [Hervorhebung durch den Verfasser])

(3) In $G(R)$ kann es keine isolierten Knoten geben, da nur solche RDF-Terme Knoten sein können, welche an Subjekt- oder Objekt-Position in Tripeln des Graphen vorkommen.

(4) Ein Tripel $(s, p, o) \in R$ identifiziert genau eine Kante in $G(R)$ vom Knoten s zum Knoten o, welche mit p markiert ist.

(5) p identifiziert die dem Tripel $(s, p, o) \in R$ zugeordnete Kante in $G(R)$ *nicht*. p repräsentiert eine binäre Relation (Property), das Paar (s, o) enthält. p kann somit nur als Markierung der Kante verstanden werden, welche den Startknoten s mit dem Zielknoten o verbindet.

(6) Der RDF-Term p eines Tripels $(s, p, o) \in R$ kann jedoch einen *Knoten* im Graph $G(R)$ identifizieren, sofern er als Subjekt oder Objekt eines Tripels in R auftritt.

(7) Mehrfachkanten sind in $G(R)$ möglich:
Zwei Tripel (s, p_1, o) und (s, p_2, o), mit $p_1 \neq p_2$, beschreiben zwei unterschiedliche Kanten zwischen dem selben Subjekt-Objekt-Paar.

Aus Punkt (2) folgt aus unserer Sicht, dass *jedes* Tripel-Subjekt und -Objekt *genau einem* Knoten entspricht. Dies ist unabhängig davon, ob es sich dabei um IRIs, Blank Nodes oder Literale handelt. In dieser Weise gehen auch alle in diesem Bericht vorgestellten Transformationen vor, welche Literale auf Knoten abbilden. Interessanterweise behandelt das W3C Literale in seinem RDF-Validierungsservice [1] anders. Dieser Service validiert die Beschreibung eines RDF-Graphen, welcher im RDF/XML-Format gegebenen ist. Als Ergebnis wird auch eine Graphdarstellung als Visualisierung generiert. Während IRIs und Blank Nodes in dieser Visualisierung genau einem Knoten zugeordnet werden und somit als eindeutig verstanden werden, egal in wie vielen Tripeln sie vorkommen, weicht die Behandlung der Literale hiervon ab. Es gibt genau so viele Literale repräsentierende Knoten, wie es Literal-Tripel gibt. D.h. syntaktisch identische Literale in verschiedenen Tripeln führen zu je einem zugeordneten Knoten per Tripel. Auch wenn dies nicht klar aus den Hinweisen der Recommendations folgt, ist eine solche Lesart aus semantischen Überlegungen nachvollziehbar: Ein Literal steht für einen Attributwert eines Individuums, der durch den RDF-Graphen beschriebenen Domäne. Es kann somit als genau diesem Individuum zugeordnet betrachtet werden. Diese Sichtweise spiegelt sich in später betrachteten Transformationen von Hartig (Abschn. 7.1) und Thakkar (Abschn. 7.2) in Labeled-Property-Graphen wieder. Dort werden die Literal-Tripel einer Ressource auf Propertys des dieser Ressource zugeordneten Knotens abgebildet. Verschiedene Literal-Tripel führen zu unterschiedlichen Propertys.

Weitere Aspekte, welche z.B. benannte RDF-Graphen oder Datasets betreffen, betrachten wir hier nicht, da im Weiteren dieses Berichts nicht relevant.

[1] https://www.w3.org/RDF/Validator/

5 Transformation von RDF-Graphen in Node Pair Labeled-Graphen

In diesem ersten Kapitel, welches eine Transformation betrachtet, verwenden wir drei Darstellungsformen, die wir auch für die im Weiteren dieses Berichts präsentierten Transformationen beibehalten werden.

- Eine Definition des dem RDF-Graphen zugeordneten formalen Graphen.
- Eine tabellarische Darstellung der Transformation der Komponenten des RDF-Graphen in die Strukturelemente des formalen Graphen.
- Eine Veranschaulichung anhand unseres Beispiel-RDF-Graphen R^{Pets} aus Kap. 2.

In diesem Bericht werden wir zudem alle Transformationen auf unseren *vollständigen* Beispiel-RDF-Graphen R^{Pets} anwenden. Grund hierfür ist, dass wir nur Transformationen betrachten wollen, welche keine über das RDF-Vokabular des Namespace rdf hinausgehenden spezifischen strukturellen oder semantischen Annahmen über die im Graphen verwendeten Vokabularelemente treffen. Ansätze, welche spezifische Transformationen z.B. für schemabeschreibende RDF-Graphen auf Basis des RDFS-Vokabulars (wie z.B. $R^{Pets-Schema}$) vorschlagen, werden nicht betrachtet.

Folgende Def. 5.1 erfasst die formale Definition des einem gegebenen RDF-Graphen R zugeordneten NPL-Graphen $NPLG(R)$. Die Struktur des zugeordneten Graphen ergibt sich in einfacher Weise aus den im vorangehenden Abschnitt zusammengefassten Sachverhalten, bei enger Orientierung an den Aussagen des Standards. Tab. 5.1 zeigt die Transformation vorab in schematischer Weise.

Tab. 5.1: Transformation der Komponenten eines RDF-Graphen R in den Node-Pair-Labeled-Graphen $NPLG(R)$ gemäß Def. 5.1.

	Def. 5.1	Komponenten aus RDF-Graph R	Zugeordnete Komponenten in $NPLG(R)$
(a)	(m-1) (f-1)	$r \in SUBJ(R) \cup OBJ(R)$	r
(b)	(m-3)	$(s,p,o) \in R$	s —p→ o

Definition 5.1.

Sei R ein RDF-Graph.

Der Node-Pair-Labeled Graph $NPLG(R) = (V, E, \mathcal{L}_E, id_V)$ heisst *der dem RDF-Graph R zugeordnete NPL-Graph*, wenn für ihn ergänzend zu Def.3.2 gilt:

(m-1) $|V| = |SubjT(R) \cup ObjT(R)|$
(Die Anzahl der Knoten entspricht der Anzahl der Tripel-Subjekte und -Objekte.)

(m-2) $\mathcal{L}_E = PredT(R)$
(Alle Tripel-Prädikate sind Kanten-Label.)

(f-1) $idV : V \mapsto (SubjT(R) \cup ObjT(R))$.
(Alle Tripel-Subjekte und -Objekte sind Identifikatoren der Knoten.)

(m-3) $(v_s, p, v_o) \in E$ gdw. $(id_V(v_s), p, id_V(v_o)) \in R$.
(Jedem Tripel (s, p, o) entspricht genau eine mit dem Tripel-Prädikat gelabelte Kante, deren Startknoten durch das Tripel-Subjekt sowie deren Endknoten durch das Tripel-Objekt identifiziert wird.)

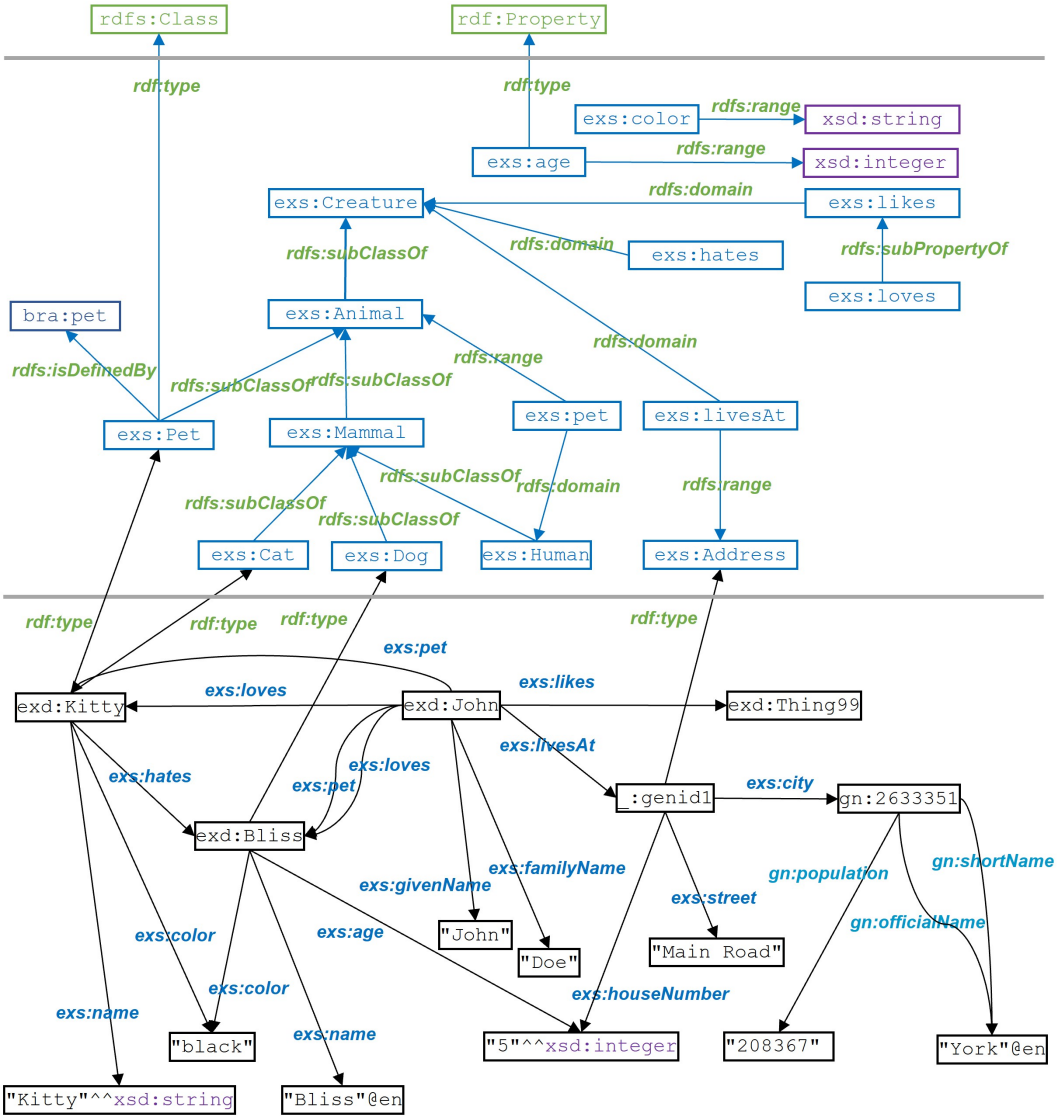

Abb. 5.1: Visualisierung des dem Beispiel-RDF-Graphen R^{Pets} aus Kap. 2 , S. 13 zugeordneten Node-Pair-Labeled-Graphen $NPLG(R^{Pets})$ gemäß Def. 5.1.

Jedes Tripel des RDF-Graphen wird hier direkt in genau eine Kante übertragen, wobei Subjekt und Objekt des Tripels die Identifikatoren der durch diese Kante verbundenen Knoten darstellen. Das Prädikat des Tripels ist das Label der zugeordneten Kante.

In dieser Formalisierung werden die Kanten (bzw. die sie konstutierenden Knotenpaare) durch die zugeordneten Label typisiert. Um dagegen Knoten explizit zu typisieren, müssen diese per Kante mit Label `rdf:type` mit einem Knoten der ihren Typ repräsentiert verknüpft werden.

Die Formalisierung von RDF-Graphen mittels NPL-Graphen bildet somit sehr eng die Tripelmenge ab.

Abb. 5.1 visualisiert den formalen RDF-Graphen $NPLG(R^{Pets})$, der unserem Beispiel-RDF-Graphen aus R^{Pets} aus Kap. 2, S. 13, gemäß Def. 5.1 zugeordnet ist. In der Darstellung haben wir ergänzend die verschiedenen Modellierungsebenen - Daten- bzw. Instanzenebene der Domäne, domänenspezifische Schema-Beschreibung und epistemologische Konzepte - abgegrenzt[1]. Die Brücken zwischen diesen Ebenen werden jeweils durch Typisierungs-Tripel etabliert. Zu beachten ist, dass diese Transformation rein strukturell ist. Sie nutzt kein zugrunde gelegtes Schlussfolgerungsregimes, wie z.B. RDF- oder RDFS-Entailment, aus.

Bonifati et al. (2018) formalisiert RDF-Graphen zwar nicht, aber er ordnet sie in seine Graphhierarchie (S.12, Fig.2.7) als gerichtete Multi-Graphen mit genau einem Label pro Kante ein. Unsere Definition ist hierzu konform.

[1] bzgl. der Farbgebung siehe auch Abb. 3.1

6 Transformation von RDF-Graphen in Labeled-Graphen

In diesem Kapitel betrachten wir drei Varianten der Formalisierung von RDF-Graphen als Graphen, welche mit Labeln markiert werden können.

6.1 Transformation in Labeled-Graphen als Modifikation der Node Pair Labeled-Graph-Transformation

Um formal für Erweiterungen um weitere Markierungen besser vorbereitet und zu anderen Ansätzen besser anschlussfähig zu sein, übertragen wir den obigen NPL-Graph-Ansatz in naheliegender Weise auf eine Repräsentation mittels L-Graphen. Die einzige Änderung ist hier, dass Kanten nun First-Class-Strukturobjekte und somit direkt identifizierbar sind. Unverändert werden die Tripel-Prädikate zu Kanten-Labeln und die Tripel-Subjekte und -Objekte zu Knoten-Identifikatoren. Tab. 6.1 stellt die angepasste Transformation schematisch dar, Def. 6.1 beschreibt formal den zugeordneten L-Graphen. Abb. 6.1, S. 40, zeigt das Ergebnis dieser Transformation für unseren RDF-Graphen R^{Pets}. Der einzige Unterschied zur NPLG-Variante in Abb. 5.1, S. 31, ist, dass die Kanten jetzt einzeln direkt identifiziert werden und nicht indirekt über die Knoten.

Tab. 6.1: Transformation der Komponenten eines RDF-Graphen R in den Labeled-Graphen $LG(R)$ gemäß Def. 6.1.

	Def. 6.1	Komponenten aus RDF-Graph R	Zugeordnete Komponenten in $LG(R)$
(a)	(m-1) (f-1)	$r \in SUBJ(R) \cup OBJ(R)$	r
(b)	(nb-1)	$(s,p,o) \in R$	s —eid| p→ o
Ergebnistyp von $LG(R)$: (0,1)			

Definition 6.1. (Adaption Def. 5.1)

Sei R ein RDF-Graph

Der L-Graph $LG(R) = (V, E, \mathcal{L}_V, \mathcal{L}_E, id_V, id_E, vrt, lbl_V, lbl_E)$ heisst *der dem RDF-Graph R zugeordnete L-Graph*, wenn für ihn ergänzend zu Def. 3.3 gilt:

(m-1) $|V| = |SubjT(R) \cup ObjT(R)|$
(Die Anzahl der Knoten entspricht der Anzahl der Tripel-Subjekte und -Objekte.)

(m-2) $|E| = |R|$
(Die Anzahl der Kanten entspricht der Anzahl der Tripel.)

(m-3) $\mathcal{L}_V = \emptyset$
(Es gibt keine Knotenlabel.)

(m-4) $\mathcal{L}_E = PredT(R)$
(Alle Tripel-Prädikate können Kantenlabel sein.)

(f-1) $idV : V \mapsto (SubjT(R) \cup ObjT(R))$.
(Alle Tripel-Subjekte und -Objekte sind Identifikatoren der Knoten.)

(f-2) $idE : dom(idE) = \emptyset$.
(Kanten erhalten keine spezifischen Identifikatoren.)

(f-4) lbl_V: $lbl_V(v) = \emptyset$, $v \in V$.
(Alle Knoten sind ungelabelt).

(f-5) lbl_E: $lbl_E(e) = \{l\}, e \in E, l \in PredT(R)$.
(Jeder Kante ist eine IRI zugewiesen, welche als Tripel-Prädikat vorkommt.)

(nb-1) Die Funktionen id_V, vrt und lbl_E erfüllen folgende Nebenbedingung:
$\forall (s,p,o) \in R \;\; \exists! \, e \in E :$
$$vrt(e) = (v_1, v_2) \wedge id_V(v_1) = \{s\} \wedge lbl_E(e) = \{p\} \wedge id_V(v_2) = \{o\}$$
(Jedem Tripel (s,p,o) entspricht genau eine Kante mit Label p, deren Startknoten den (eindeutigen) Identifikator s und deren Endknoten den (eindeutigen) Identifikator o hat.)

6.2 Transformation in Labeled-Graphen nach Hayes

Bereits 2004 hat Hayes den Wert einer Formalisierung von RDF-Graphen als formale Graphen erkannt (S.56):

> *"Representing RDF by standard graphs could have several other advantages by reducing problems to well-studied topics from graph-theory."*

Explizit nennt er neben dem Aspekt der Visualisierung die Themen Entailment von RDF-Graphen, für welche Erkenntnisse bzgl. Graph-Isomorphismen genutzt werden könnten, Minimierung von RDF-Graphen, das Auffinden semantischer Verknüpfungen zwischen Ressourcen auf Basis existierender Pfadverbindungen sowie die Identifizierung von Clustern und die Berechnung anderer strukturellen Dekompositionen.(Hayes (2004),S.55/56)

Mit Labeled-Property-Graphdatenbanken stehen inzwischen genau solche Systeme zur Verfügung, die zahlreiche der genannten Analysemöglichkeiten bereitstellen und dies zudem für damals noch nicht vorstellbare Graphgrößen.

Tab. 6.2: Transformation der Komponenten eines RDF-Graphen R in den Labeled-Graphen $LG^{Hayes}(R)$ gemäß Def. 6.2.

	Def. 6.2	Komponenten aus RDF-Graph R	Zugeordnete Komponenten in $LG^{Hayes}(R)$
(a)	(m-1) (f-1)	$r \in SubjT(R) \cup ObjT(R)$	r map(r)
(b)	(f-2) (nb-2)	$(s,p,o) \in R$	s map(s) — eid | p — o map(o)
Ergebnistyp von $LG^{Hayes}(R)$: (1,1)			

In Kap. 5 seiner Arbeit formalisiert Hayes RDF-Graphen als gerichtete, markierte Multi-Graphen. Sein Ansatz ähnelt sehr unserer vorangehenden Formalisierung. Jedes Tripel wird auf genau eine Kante

abgebildet. Allerdings verwendet er *alle* in R vorkommenden RDF-Terme als *Label* von Knoten bzw. Kanten. Tab. 6.2 und Def. 6.2 bilden seinen Ansatz in unsere Notation ab.

Definition 6.2. (vgl. Hayes (2004), Def. 11, S. 63)

Sei R ein RDF-Graph.

Der L-Graph $LG^{Hayes}(R) = (V, E, \mathcal{L}_V, \mathcal{L}_E, id_V, id_E, vrt, lbl_V, lbl_E)$ heisst *der dem RDF-Graph R nach Hayes zugeordnete L-Graph*, wenn für ihn ergänzend zu Def. 3.3 gilt:

(m-1) $|V| = |SubjT(R) \cup ObjT(R)|$
$map : SubjT(R) \cup ObjT(R) \rightarrow V$ sei eine totale bijektive Funktion.

(m-2) $|E| = |R|$

(m-3) $\mathcal{L}_V = SubT(R) \cup ObjT(R)$;

(m-4) $\mathcal{L}_E = PredT(R)$

(f-1) lbl_V: bijektiv mit $lbl_V(v) = \{l\}, v \in V, l \in (SubjT(R) \cup ObjT(R))$.
(Jeder Knoten ist eindeutig mit einer IRI, einem Blank Node oder einem Literal als Label markiert.)

(f-2) lbl_E: $lbl_E(e) = \{l\}, e \in E, l \in PredT(R)$.
(Jeder Kante ist eine IRI zugewiesen, welche als Tripel-Prädikat vorkommt.)

(nb-2) Die Funktionen lbl_V und lbl_E erfüllen folgende Nebenbedingung:
$\forall (s,p,o) \in R \ \exists! \, e \in E : \ vrt(e) = (v_1, v_2)$
mit $lbl_V(v_1) = \{s\} \wedge lbl_E(e) = \{p\} \wedge lbl_V(v_2) = \{o\}$
(Jedem Tripel (s,p,o) entspricht genau eine Kante mit Label p, deren Startknoten das (eindeutige) Label s und deren Endknoten das (eindeutige) Label o hat.)

Der von Hayes gewählte, auf den ersten Blick sehr klare Ansatz, alle Tripelkonstituenten als Label zu verwenden, hat jedoch den Nachteil (worauf Hayes selbst hinweist), dass damit die Label des zugeordneten Graphen unterschiedliche Bedeutungen besitzen: Knoten-Label benamen Individuen einer Klasse, während Kanten-Label den Typ der Beziehung zwischen diesen Individuen repräsentieren. Diese uneinheitliche Labelsemantik führt zu dem Effekt, dass eine IRI, die eine Property repräsentiert, sowohl Label von Kanten, wie auch eines Knotens sein kann, wie in Abb. 6.2, S. 41, zu sehen ist (z.B. `exs:pet`, `exs:age`). In den zuvor betrachteten Varianten (siehe NPL-Graph, Abb. 5.1, S. 31, und L-Graph, Abb. 6.1, S. 40) tritt dagegen eine solche IRI, z.B. `exs:age`, als Kanten*label* und als Knoten*identifikator* auf. Dieses ist aus unserer Sicht semantisch adäquater, denn: Der Knoten, der durch die Property-IRI identifiziert wird, ist ein *Individuum* auf der nächst höheren Beschreibungsebene. So stehen z.B. die Knoten `exs:age` in den genannten Abbildungen auf der Ebene des domänenspezifischen RDFS-Schemas für eine *individuelle* Property aus der Klasse `rdf:Property` aller Propertys.

Die vorgestellten Formalisierungen von RDF-Graphen als NPL-Graphen (Def. 5.1) bzw. L-Graphen (Def. 6.1) sind diesbezüglich semantisch konsistenter und aus unserer Sicht damit der Hayes-Variante vorzuziehen.

6.3 Transformation in Labeled-Graphen nach Thakkar

Thakkar präsentiert verschiedene Transformationen von RDF-Graphen in das Labeled-Property-Graph-Modell (siehe Thakkar (2021), Kap. 4[1]), welche wir auch später in Abschn. 7.2 betrachten werden.

[1] bereits in Angles et al. (2020b) veröffentlicht

Als Zwischenschritt transformiert er RDF-Graphen zunächst in Labeled-Graphen. Hierfür stellt er zwei Ansätze vor:

- Der erste dient der Transformation von schemalosen RDF-Graphen, also reinen Datenbeschreibungen (vgl. Thakkar, Def. 2 *RDF Graph*, S.64). Unser Graph $R^{Pets-Inst}$ ist ein Beispiel hierfür. Ziel dieser Transformation ist eine Normalisierung dahingehend, dass alle Tripel-Subjekte und Literale, denen im RDF-Graphen keinen Typ zugewiesen wurde, defaultmäßig typisiert werden.
- Für RDF-Graphen, welche ein zugehöriges Datenschema auf Basis des RDFS-Vokabulars beschreiben (wie z.B. unser RDF-Graph $R^{Pets-Schema}$), schlägt Thakkar einen eigenen zweiten Ansatz vor (Def. 3 *RDF Graph Schema*, S.68). Diese zweite Transformation betrachten wir in diesem Bericht nicht näher, weil wir den Aspekt der Transformationen von Schemata nicht behandeln. Wir werden vielmehr die von Hayes vorgeschlagene erste Transformation auf unseren vollständigen RDF-Graphen R^{Pets} anwenden (siehe hierzu auch die Eingangsbemerkungen zu Kap. 5).

Tab. 6.3: Transformation der Komponenten eines RDF-Graphen R in den Labeled-Graphen $LG^{mod-Thakkar}(R)$ gemäß Def. 6.3

	Def. 6.3	Komponenten aus RDF-Graph R $IRI_{RootClass} \in \mathcal{IRI}$	Zugeordnete Komponenten in $LG^{mod\text{-}Thakkar}(R)$
(a)	(m-1) (f-1)	$r \in \big(IRI(R) \cup BN(R)\big)$ $\cap\ (SubjT(R) \cup ObjT(R \backslash R_{TYPES}))$	*(siehe b,c)* r
(b1)	(m-1) (f-1) (f-4)	$r \in \big(IRI(R) \cup BN(R)\big)$ $\cap\ (SubjT(R) \cup ObjT(R \backslash R_{TYPES}))$ $R_{TYPE}(r) = \{t_1, \ldots, t_n\} \neq \emptyset$ mit $t_i = (r, \texttt{rdf:type}, c_i),\ 1 \leq i \leq n$	$c_1, \ldots, c_n$ r
(b2)	(m-1) (f-1) (f-4)	$r \in \big(IRI(R) \cup BN(R)\big)$ $\cap\ (SubjT(R) \cup ObjT(R \backslash R_{TYPES}))$ $R_{TYPE}(r) = \emptyset$	$IRI_{RootClass}$ r
(c1)	(m-1) (f-1) (f-4)	l = `value` $\in LIT(R)$ l einfaches Literal	*xsd:string* `value`
(c2)	(m-1) (f-1) (f-4)	l = `"value"^^datatypeIRI` $\in LIT(R)$ l Literal mit Datentypangabe	*datatypeIRI* `"value"^^datatypeIRI`
(c3)	(m-1) (f-1) (f-4)	l = `"value"@lang` $\in LIT(R)$ l Literal mit Language-Tag	*rdf:langString* `"value"@lang`
(d)	(nb-1)	$(s, p, o) \in R \backslash R_{TYPES}$	*(siehe b,c)* s — `eid` | **p** → *(siehe b,c)* o
Ergebnistyp von $LG^{mod\text{-}Thakkar}(R)$: *(+,1)* [gem. orig. Thakkar(2021): *(1,1)*]			

Thakkars Grundideen zur Repräsentation eines datenbeschreibenden RDF-Graphen R^{Data} als L-Graph, sind:

- Alle Resourcen, die Tripel-Subjekte und -Objekte sind, identifizieren Knoten.
- Existiert für ein Tripel-Subjekt ein Typisierungs-Tripel, dann wird der zugeordnete Knoten mit der entsprechenden Typ-IRI gelabelt, andernfalls wird rdfs:Resource als Default-Label verwendet.

- Ist ein Literal-Term mit einer Datentyp-IRI versehen, wird diese als Knotenlabel zugewiesen, alle einfachen Literal-Terme erhalten den Datentyp xsd:string als Default-Label.

Folgende Aspekte sind bzgl. Thakkars Vorgehen hervorzuheben:

- RDF-Terme im Inputgraphen R^{Data} dürfen max. ein Typisierungs-Tripel besitzen.
- Die Transformation ist nicht vollständig schemalos, in dem Sinne, dass sie sich implizit auf die RDFS-Semantik stützt. Denn die Verwendung von rdfs:Resource als Default-Typ für Tripel-Subjekte entspricht der Anwendung der RDFS-Entailment-Regel *rdfs4a* (vgl. Hayes and Patel-Schneider (2014), Abschn. 9.2.1).
- Im Ergebnisgraph ist allen Knoten genau ein Label zugewiesen, welches den Typ der mit dem Knoten assoziierten Ressource repräsentiert.

Thakkars Ansatz für datenbeschreibende RDF-Graphen erweitern wir um die folgenden Punkte und formalisieren diese Modifikation auf Basis unserer Notation (siehe Tab. 6.3 und Def. 6.3):

- Die Restriktion auf maximal ein Typisierungs-Tripel je vorkommendem RDF-Term entfällt. Thakkars Einschränkung ist sinnvoll, wenn man in der modellierten Domäne davon ausgehen kann, dass es genau einen speziellsten Typ für jedes Domänenobjekt gibt. Geht man von einer Klassenhierarchie aus, dürften damit keine Typisierungs-Tripel zu Supertypen explizit im Inputgraphen vorhanden ein. Zudem hat man es oft mit Multihierarchien zu tun, welche die Zuordnung eines einzigen speziellsten Typs nicht immer erlauben. In unserem Beispiel-RDF-Graph R^{Pets} deuten wir dies anhand der Ressource exd:Kitty an. Dieser sind die Typen exd:Cat und exd:Pet zugeordnet, welche beide Blätter der beschriebenen Klassenhierarchie sind.
- Die Transformation wird mit einem Default-Typ parametrisiert. Diesen erhalten sowohl alle Tripel-Subjekte wie auch alle Nicht-Literal-Tripel-Objekte bei fehlenden expliziten Typisierungs-Tripeln als Label zugewiesen.[2]
  Durch diese Parametrisierung kann die Transformation flexibel an semantische Grundannahmen angepasst werden. Spezifiziert z.B. eine OWL-Ontologie die im RDF-Graphen verwendete Terminologie, so kann owl:Thing als Default-Typ angegeben werden.
- Literal-Termen mit Language-Tag wird gemäß Cyganiak et al. (2014), Abschn. 3.3, der Datentyp rdf:langString als Default-Label zugeordnet.
- Typisierungs-Tripel werden nicht zu Kanten transformiert.

Definition 6.3. (vgl. Thakkar (2021), Def. 2 *RDF Graph*, S.64)

Sei R ein RDF-Graph und $IRI_{RootClass}$ eine IRI.

Der L-Graph $LG^{mod-Thakkar}(R, IRI_{RootClass}) = (V, E, \mathcal{L}_V, \mathcal{L}_E, id_V, id_E, vrt, lbl_V, lbl_E)$ heisst *der dem RDF-Graph R nach Thakkar modifiziert zugeordnete L-Graph*, wenn für ihn ergänzend zu Def. 3.3 gilt:

[2] Die Einbeziehung der Nicht-Literal-Tripel-Objekte bedeutet im Zusammenhang mit dem Default-Typ rdfs:Resource die Umsetzung der RDFS-Entailment-Regel *rdfs4b*. Wir bleiben so konsistent zur originalen Vorgehensweise Thakkars.

(m-1) $|V| = |SubjT(R) \cup ObjT(R \setminus R_{TYPES})|$
(Die Anzahl der Knoten entspricht der Anzahl der Tripel-Subjekte und -Objekte, ohne Objekte von Typisierungs-Tripeln, welche nicht selbst auch Subjekt sind. Dies vermeidet isolierte Typ-Knoten[a]. Insbesondere gibt es für auschließlich als Prädikat verwendete IRIs keine Knoten.)

[a] Wir folgen hier Thakkars Ansatz. Er ist begründet in seiner Vorstellung reine Datengraphen zu transformieren. Dann ist es schlüssig isoliert entstehende Typ-Knoten zu vermeiden. Transformiert man jedoch einen RDF-Graphen, welcher auch Schema-Tripel enthält, dann wäre es sinnvoll, dass alle verwendeten Typen durch Knoten repräsentiert sind.

(m-2) $|E| = |R \setminus R_{TYPE}|$
(Die Anzahl der Kanten entspricht der Anzahl der Tripel in R ohne die Typisierungs-Tripel.)

(m-3) $\mathcal{L}_V = \{o \mid \exists\, t \in R:\ pred(t) = \texttt{rdf:type} \wedge obj(t) = o\}$
$\cup\, \{dtm(\texttt{datatypeIRI}) \mid$
$\exists\, t \in R:\ obj(t) \in LIT(R) \wedge obj(t)\,\text{enthält}\ \texttt{\^\^datatypeIRI}\}$
$\cup\, \{IRI_{RootClass}\}$
$\cup\, \{dtm(\texttt{xsd:string})\} \cup \{dtm(\texttt{rdf:langString})\}$
(Alle RDF-Terme, welche zur Typisierung in R verwendet werden (im Rahmen von Typisierungs-Tripeln oder typisierten Literalen) können Knotenlabel sein, sowie die IRI der gegebenen Wurzelklasse als Defaulttyp.)

(m-4) $\mathcal{L}_E = PredT(R)$
(Alle Tripel-Prädikate können Kantenlabel sein.)

(f-1) $idV : V \mapsto (SubjT(R) \cup ObjT(R \setminus R_{TYPES}))$.
(Alle Tripel-Subjekte und -Objekte, für welche Knoten erzeugt werden, sind Identifikatoren dieser Knoten. Zur Sicherstellung der Eindeutigkeit der Literale müssen diese inkl. Datentypangabe bzw. Language-Tag (falls vorhanden) als Identifikatoren verwendet werden.)

(f-2) $idE : dom(idE) = \emptyset$.
(Kanten erhalten keine spezifischen Identifikatoren.)

(f-4) lbl_V:
Für $v \in V$ mit $idV(v) \in IRI(R) \cup BN(R)$:
$$lbl_V(v) = \begin{cases} ObjT(R_{TYPE}(idV(v))) & \text{falls nicht leer} \\ \{IRI_{RootClass}\} & \text{sonst} \end{cases}$$
Für $v \in V$ mit $idV(v) \in LIT(R)$:
$$lbl_V(v) = \begin{cases} \texttt{datatypeIRI} & \text{falls}\, idV(v)\,\text{enthält}\ \texttt{\^\^datatypeIRI} \\ \texttt{xsd:string} & \text{falls}\, idV(v)\,\text{einfaches Literal} \\ \texttt{rdf:langString} & \text{falls}\, idV(v)\,\text{Literal mit Language} - \text{Tag} \end{cases}$$
(Alle Knoten, deren Identifikator kein Literal ist, sind mit den Objekt-Termen der Typisierungs-Tripel ihres Identifikators gelabelt, falls mindestens ein Typisierungs-Tripel vorhanden. Andernfalls wird die IRI der gegebenen Wurzelklasse als Default zugewiesen.
Alle Knoten, deren Identifikator ein Literal ist, erhalten ein Label abhängig von der Art dieses Literals. Bei Literalen mit expliziter Datentyp-IRI, ist diese IRI das Typ-Label. Einfache Literale bzw. Literale mit Language-Tag erhalten die Default-Typen xsd:string bzw. rdf:langString.)

(f-5) lbl_E: $lbl_E(e) = \{l\}, e \in E, l \in PredT(R \setminus R_{Types})$.
(Jeder Kante ist eine IRI ungleich rdf:type zugewiesen, welche als Tripel-Prädikat vorkommt.)

(nb-1) Die Funktionen id_V, vrt und lbl_E erfüllen folgende Nebenbedingung:
$\forall (s,p,o) \in R \setminus R_{TYPE}\ \exists!\, e \in E:$
$$vrt(e) = (v_1, v_2) \wedge id_V(v_1) = \{s\} \wedge lbl_E(e) = \{p\} \wedge id_V(v_2) = \{o\}$$
(Jedem Tripel (s,p,o) entspricht genau eine Kante mit Label p, deren Startknoten den (eindeutigen) Identifikator s und deren Endknoten den (eindeutigen) Identifikator o hat.)

Abb. 6.3, S. 42, visualisiert das Ergebnis der Transformation gem. Def. 6.3 für unseren Beispiel-RDF-Graphen. Insbesondere haben wir - wie oben schon kommentiert - auch das in diesen RDF-Graphen eingebettete RDFS-Schema mit transfomiert. Man erkennt sehr gut, dass durch diese Transformation die Beschreibungsebenen topologisch voneinander abgetrennt werden. Dies gilt sowohl mit Blick auf die Domänendaten und das die Domänenterminologie beschreibende RDFS-Schema, aber auch für das RDFS-Schema und die Meta-Ebene mit den epistemologischen Vokabularelementen von RDFS. Die entstehende topologische Trennung führt dazu, dass eine Pfadnavigation von Instanz-Knoten zu für sie relevanten Schemaknoten nicht möglich ist. Z.B. kann man ausgehend vom Knoten exd:Kitty im Graphen $LG^{mod-Thakkar}(R^{Pets}, \texttt{rdfs:Resource})$ nicht zu den Knoten exs:Cat, exs:Pet, exs:Mammal bzw. exs:Animal navigieren. Vielmehr müssten die Label des Knotens exd:Kitty abgefragt und als Einstiegsknoten in die Klassenhierarchie verwendet werden.

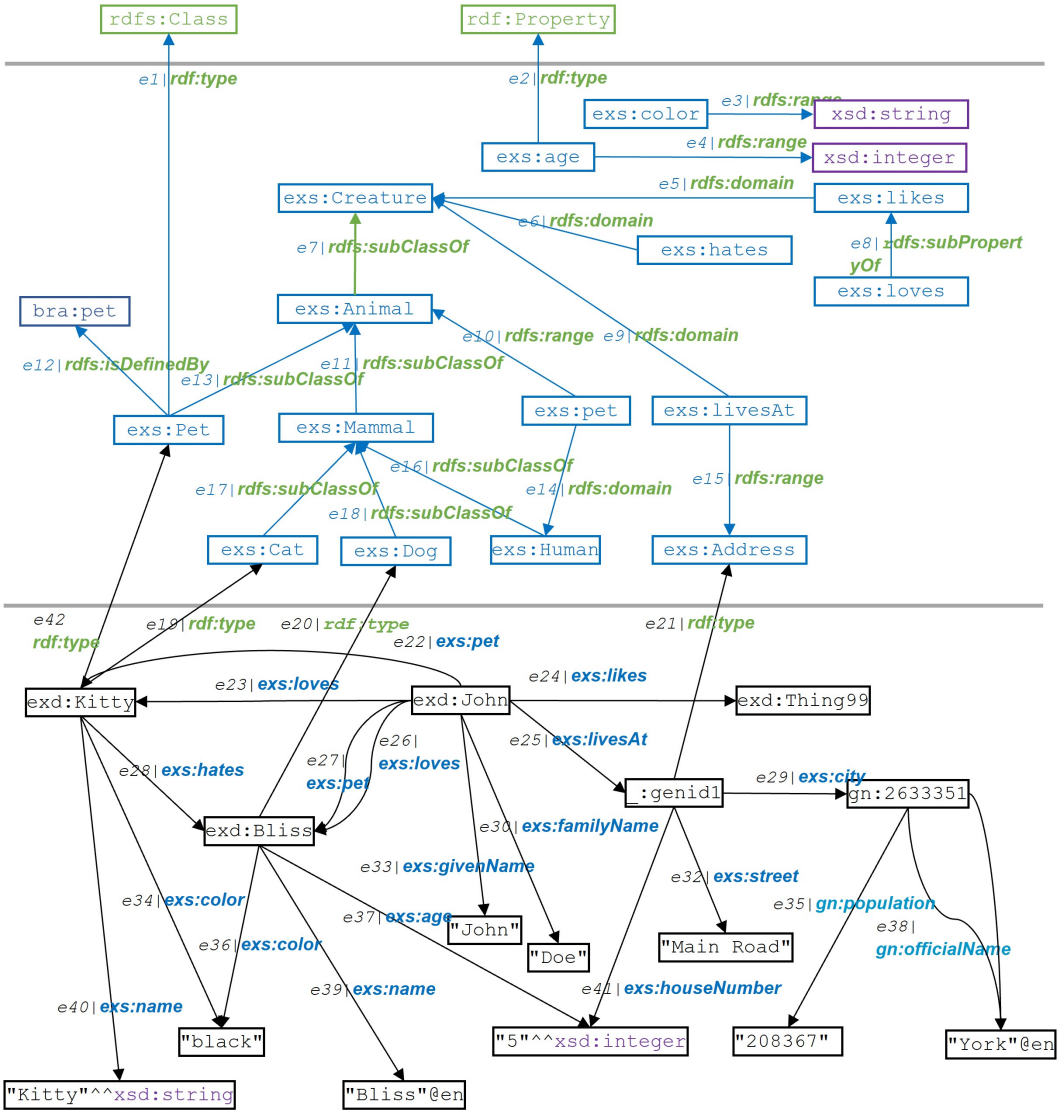

Abb. 6.1: Visualisierung des dem Beispiel-RDF-Graphen R^{Pets} aus Kap. 2 , S. 13 zugeordneten Labeled-Graphen $LG(R^{Pets})$ gemäß Def. 6.1.

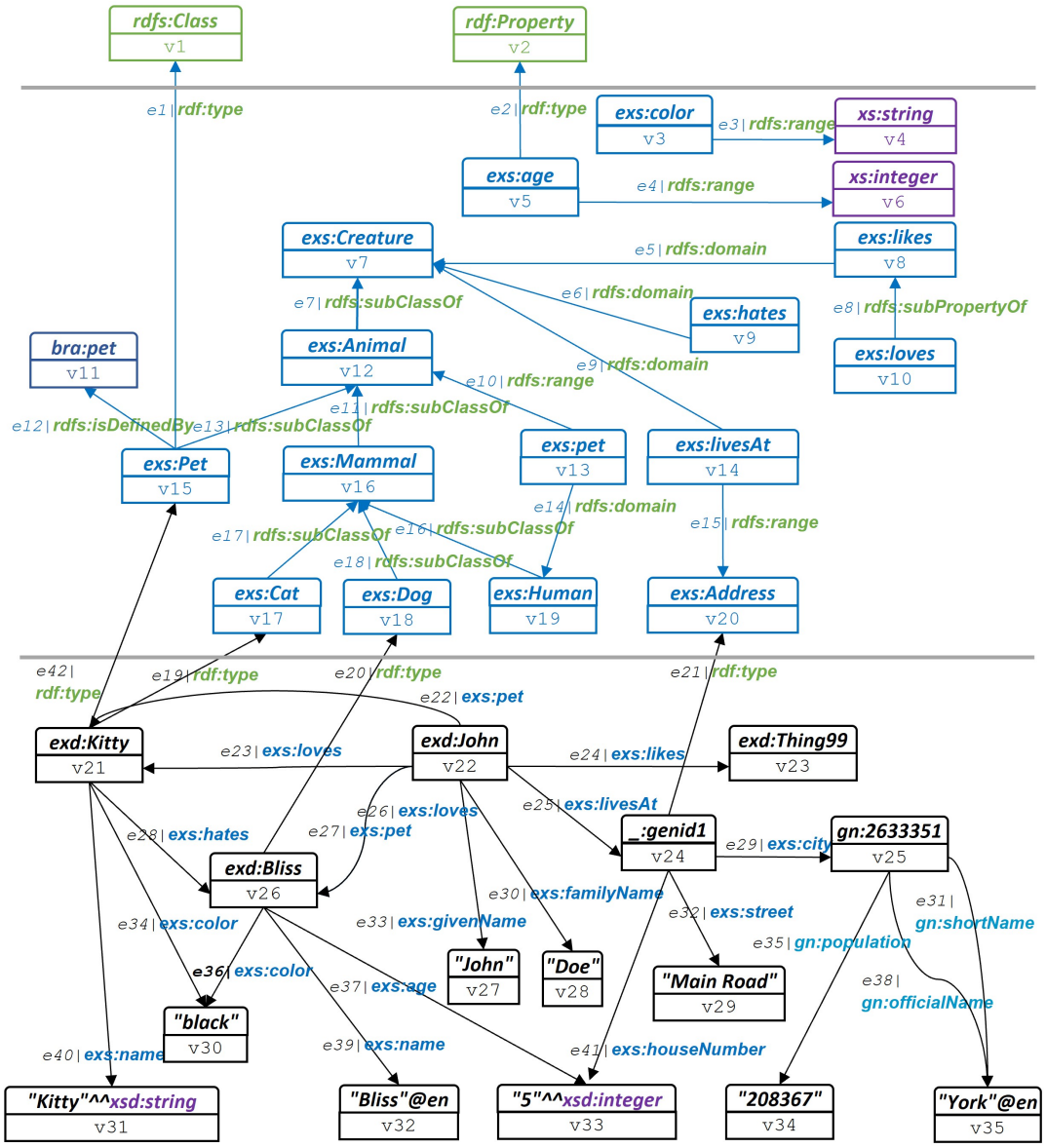

Abb. 6.2: Visualisierung des dem Beispiel-RDF-Graphen R^{Pets} aus Kap. 2, S. 13 zugeordneten Labeled-Graphen $LG^{Hayes}(R^{Pets})$ gemäß Def. 6.2

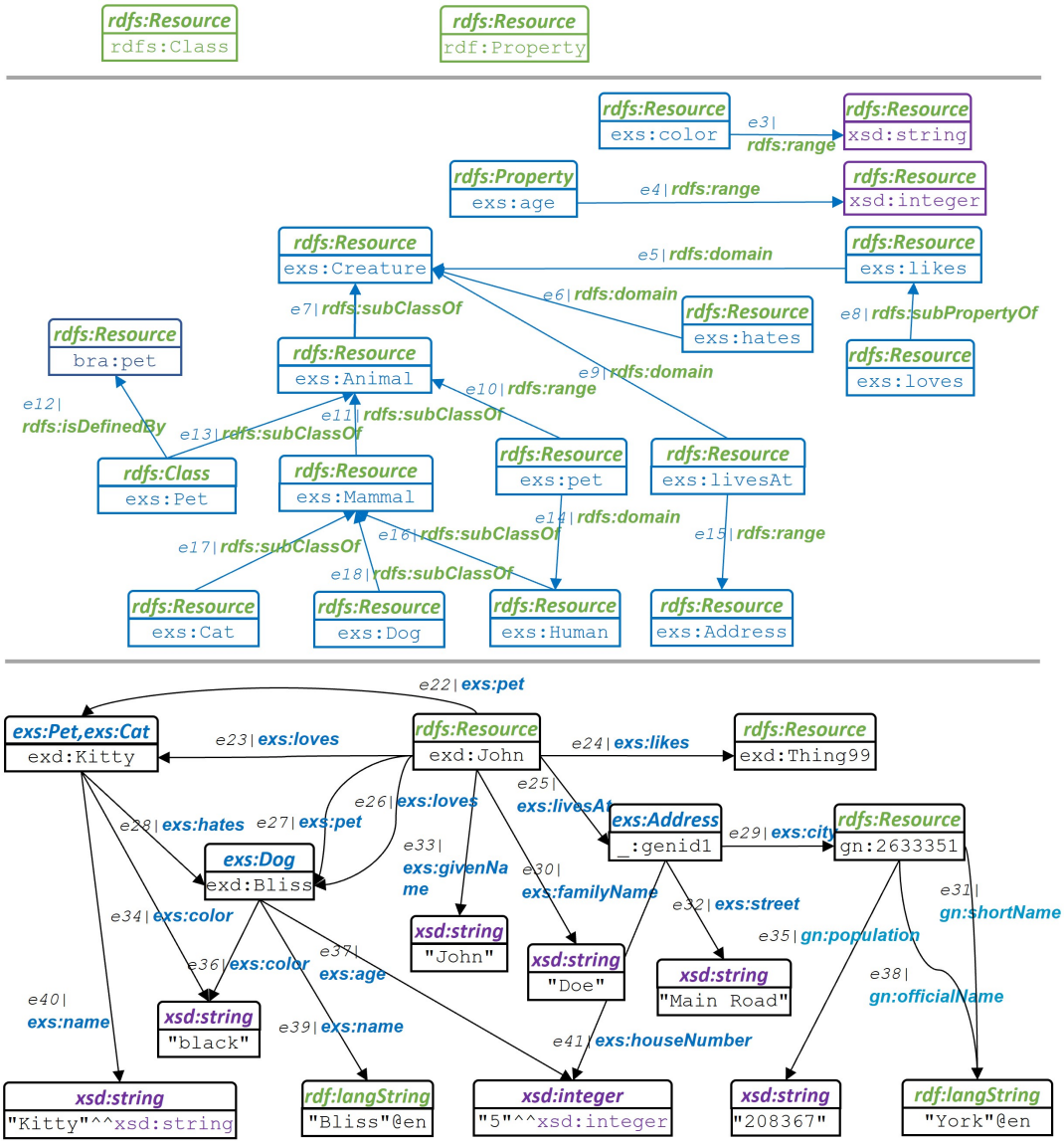

Abb. 6.3: Visualisierung des dem Beispiel-RDF-Graphen R^{Pets} aus Kap. 2 , S. 13 zugeordneten Labeled-Graphen $LG^{mod-Thakkar}(R^{Pets}, \texttt{rdfs:Resource})$ gemäß Def. 6.3.

7 Transformation von RDF-Graphen in Labeled-Property-Graphen

Ein wesentlicher Aspekt hinsichtlich der Interoperabilität von RDF- und Labeled-Property-Graphdatenbanken ist die Frage, inwieweit ihre Datenmodelle ineinander transformierbar sind.

Bereits in Thakkars Formalisierung von RDF-Graphen als Labeled-Graphen (siehe Abschn.6.3) werden die Label zur Repräsentation von Typzugehörigkeiten verwendet. Diese Nutzung von Labeln ist auch in Labeled-Property-Graphdatenbanken sehr gebräuchlich[1]. Ergänzend ist nun auch die übliche Verwendung von Propertys von Interesse.

Zhao et al. (2018) skizziert in Abschn. 3.1 die Bedeutungen der Strukturelemente von Labeled-Property-Graphen wie folgt:

- *"Labels are one of the foundational elements of LPG data model. Both vertices and edges have labels [...] The vertices labels are a way to assign the roles to vertices and to categorize vertices by means of their semantic features. The vertex labels are similar to rdf:type of RDF [...]. [...] every edge has mandatorily one and only one label that represents the relationship between two connected vertices. The edge label plays a role of the unique identifier of the edge and constructs the domain graph structure."*[2]
- *"The property is the foundational mechanism of LPG model to describe the attributes of vertices (entities) and edges (relationships). Since the attributes are the intentional characteristics of an entity, object or relation, the property usually represents intrinsic or conceptual features [...]."*

In diesem Kapitel betrachten wir je zwei Transformationen von RDF-Graphen in Labeled-Property-Graphen der Autoren Hartig und Thakkar. Jeweils eine ihrer Transformationen verfolgt die Grundidee je Tripel eine Kante zu erzeugen, welche wir ja schon in einigen vorangehenden Ansätzen gesehen haben. Die anderen beiden Transformationen nutzen stärker die Möglichkeit, die im RDF-Graph beschriebenen Attributierungen von Domänenobjekten durch Propertys im zugeordneten Labeled-Property-Graph zu erfassen.

7.1 Transformationen in Labeled-Property-Graphen basierend auf Hartig

Bereits 2014 hat Hartig die Abbildung von RDF*-Graphen (siehe Hartig et al. 2021 und 2014) in Labeled-Property-Graphen betrachtet. Hierbei ging es insbesondere um die Fragestellung, inwieweit in einem RDF*-Graphen vorhandene geschachtelte Tripel im LPG-Modell ausgedrückt werden und die Abbildungen invertierbar formuliert werden können. In Hartig (2014) stellt er konkret zwei

[1] Label können dort allerdings auch ganz pragmatisch genutzt werden, um bestimmte Gruppen von Knoten in einem Index zwecks effizienterem Zugriff zusammenzufassen

[2] Die Festlegung auf maximal ein Kanten-Label würden wir bei Betrachtung der formalen Transformationen nicht treffen, da eine Standardisierung des LPG-Modells nicht vorliegt. Die genannte Einschränkung entspricht aber der üblichen Handhabung in vorhandenen LPG-Datenbanken.

Transformationen vor.

Da wir im Rahmen dieses Berichts RDF* nicht betrachten, reduzieren wir Hartigs Transformationen auf RDF-Graphen und lassen die ausschließlich für RDF*-Graphen benötigten Anteile weg[3].

Die erste Transformation erzeugt einen sog. *RDF-like Property Graph* (vgl. Hartig (2014), Def. 6, S. 12). Die Grundidee dieses Ansatzes ist es, die Vernetzungsstruktur der RDF-Terme des RDF-Graphen vollständig in der Topologie des Ergebnisgraphen zu erfassen.[4] Tab. 7.1 beschreibt diese Abbildung der Komponenten des RDF-Graphen in einen Labeled-Property-Graphen.

Tab. 7.1: Transformation der Komponenten eines RDF-Graphen R in den Labeled-Property-Graphen $LPG^{Hartig-RDF-like}(R)$ gemäß Def.7.1.

	Def. 7.1	Komponenten aus RDF-Graph R	Zugeordnete Komponenten in $LPG^{Hartig\text{-}RDF\text{-}like}(R)$
(a)	(m-1) (m-5) (nb-3)(i)	$r \in IRI(R) \cap (SubjT(R) \cup ObjT(R))$	`map(r)` kind [*dtm*(xsd:string)] = IRI IRI [*dtm*(xsd:string)] = *r*
(b)	(m-1) (m-5) (nb-3)(ii)	$r \in BN(R)$	`map(r)` kind [*dtm*(xsd:string)] = blanknode
(c)	(m-1) (m-5) (nb-3)(iv)	$r \in LIT(R)$ *r* einfaches Literal: r typisiertes Literal: $r =$ "`lval`"`^^datatypeIRI` $\in LIT(R)$ r Literal mit Language-Tag: $r =$ "`lval`"`@langtag` $\in LIT(R)$:	`map(r)` kind [*dtm*(xsd:string)] = literal SOWIE literal [*dtm*(xsd:string)] = *r* datatype [*dtm*(xsd:string)] = *dtm*(xsd:string) ODER literal [*dtm*(xsd:string)] = `lval` datatype [*dtm*(xsd:string)] = *dtm*(*datatypeIRI*) ODER value [*dtm*(xsd:string)] = `lval` type [*dtm*(xsd:string)] = `rdf:langString` lang [*dtm*(xsd:string)] = `langtag`
(d)	(m-1) (nb-4)	$(r, p, o) \in R$	`map(r)` (siehe a,b) —`eid`| *p* → `map(o)` (siehe a,b,c)
Ergebnistyp von $LPG^{Hartig\text{-}RDF\text{-}like}(R)$: $\frac{(0,1)}{(+,0)}$			

Die wesentlichen Transformationsschritte sind:

- Alle gegebenen RDF-Terme werden auf genau einen Knoten im LP-Graphen abgebildet.
- Jeder LP-Graph-Knoten erhält eine Property mit Knoten $kind$, die die Art des zugrunde liegenden RDF-Terms als Wert erhält.
- Knoten, welche IRI-Terme repräsentieren, erhalten ergänzend eine Property IRI mit dem entsprechenden IRI-String als Wert.

[3] Diese eingeschränkte Form nutzt auch Virgilio (2017) in seiner Arbeit, welche auf die Transformation von RDF-Containern fokussiert.

[4] Ein Vorteil dieses Ansatzes ist insbesondere, dass er für RDF*-Graphen (mit einigen Einschränkungen) eine verlustfreie Transformation in einen Labeled-Property-Graphen ermöglicht, da sowohl Metadatentripel für Objekt-Tripel wie für Literal-Tripel auf Propertys der zugehörigen Kanten abgebildet werden können (siehe Hartig (2014), Abschn. 3.2 bzw. 5). Für normale RDF-Graphen, auf die wir uns beschränken, entstehen dagegen keine Propertys an den Kanten.

- Knoten, welche Literal-Terme repräsentieren, erhalten ergänzend zwei Propertys $literal$ und $datatype$ sowie, falls relevant, eine dritte Property $language$ zugewiesen, welche den Literalwert, dessen Datentyp sowie bei Literalen mit Language-Tag die Sprachinformation als Value erhalten.
- Knoten, welche Blank Node-Terme repräsentieren, wird keine ergänzende Property zugeordnet. Dies trägt der Tatsache Rechnung, dass die Identifikatoren von Blank Nodes im RDF-Graphen nur lokale, nicht-persistente Bedeutung haben. D.h. bei einer Rücktransformation des Ergebnis-LP-Graphen nach RDF müssen diese Identifikatoren nicht dieselben sein wie zuvor.
- Jedem Tripel (s, p, o) des RDF-Graphen entspricht genau eine Kante, welche die s und o zugeordneten Knoten gerichtet verbindet. Diese Kante wird mit p gelabelt.

In der folgenden Definition berücksichtigen wir alle wesentlichen Aspekte von Hartigs Vorschlag und verzichten lediglich auf Punkt 7 seiner Def. 6.[5]

Definition 7.1. (vgl. Hartig (2014), Def. 6, S. 12)

Sei R ein RDF-Graph
Sei $dtm: \mathfrak{D}_{RDF} \to \mathfrak{D}$ eine Abbildung, welche Datentypen des RDF-Graphen (bzw. deren Bezeichner) in Datentypen der Property-Values abbildet (bzw. deren Bezeichner).
Es gelte $dtm(\texttt{xsd:string}) = \mathcal{S}tr$ [6].

$LPG^{Hartig-RDF-like}(R) = (V, E, \mathcal{L}_V, \mathcal{L}_E, id_V, id_E, vrt, lbl_V, lbl_E, \mathcal{K}, \mathcal{V}, P, kv_P, prp_V, prp_E)$

bezeichnet den *Labeled-Property-Graphen*, welcher dem RDF-Graphen R auf Basis von Hartigs *RDF-like Property Graph*-Transformation inkl. unserer Einschränkungen zugeordnet wird.

Für diesen gilt ergänzend zu Def. 3.5:

(m-1) $|V| = |SubjT(R) \cup ObjT(R)|$
$map: SubjT(R) \cup ObjT(R) \to V$ sei eine totale bijektive Funktion.
(Im zugeordneten LP-Graphen gibt es für alle Tripel-Subjekte- und Objekte genau einen Knoten.)

(m-2) $|E| = |R|$
(Die Anzahl der Kanten entspricht der Anzahl der Tripel in R.)

(m-3) $\mathcal{L}_V = \emptyset$
(Es gibt keine Knoten-Label.)

(m-4) $\mathcal{L}_E = Pred(R)$
(Jedes vorkommende Prädikat ist Kanten-Label.)

(m-5) $\mathcal{K} = \{kind, IRI, literal, datatype, lang\}$, mit
$dt(kind) = dt(IRI) = dt(literal) = dt(datatype) = dt(lang)$
$= dtm(\texttt{xsd:string}) = \mathcal{S}tr$
(Insgesamt werden nur fünf Keys verwendet, welche alle den Datentyp String haben.)[a]

[a] Der Datentyp für $literal$ weicht hiermit ggf. von Hartigs Ansatz ab.

(f-4) lbl_V: Sei $r \in SubjT(R) \cup ObjT(R)$, dann gilt: $lblV(r) = \emptyset$.
(Knoten besitzen keine Label.)

[5] Wir sind etwas ungenauer als Hartig in der Hinsicht, dass wir nicht zwischen den RDF-Termen und ihren String-Repräsentationen unterscheiden.

[6] $\mathcal{S}tr$ bezeichnet den Datentyp aller Strings (siehe Def. 3.1).

(nb-3) Die Funktionen kv_P und prp_V erfüllen folgende Nebenbedingung:
Sei $r \in SubjT(R) \cup ObjT(R)$. Es gilt:

(i) Falls $r \in IRI(R)$.
$\exists! p_1, p_2 \in prp_V(map(r)) : kv_P(p_1) = (kind, "IRI") \wedge kv_P(p_2) = (IRI, "r")$
(Für IRI-Terme r erhält der zugeordnete Knoten genau zwei Propertys, welche die Art des RDF-Terms r und die IRI r erfassen.)

(ii) Falls $r \in BN(R)$.
$\exists! p \in prp_V(map(r)) : kv_P(p) = (kind, "blank\ node")$
(Für Blank Node-Terme r erhält der zugeordnete Knoten genau eine Property, welche die Art des RDF-Terms r erfasst.)

(iii) Falls $r \in LIT(R)$, r einfaches Literal:
$\exists! p_1, p_2, p_3 \in prp_V(map(r)) :$
$kv_P(p_1) = (kind, "literal") \wedge kv_P(p_2) = (literal, "r")$
$\wedge kv_P(p_3) = (datatype, "\mathcal{S}tr\ ")$

(iv) Falls $r \in LIT(R)$, $r =$ `lval^^datatypeIRI` typisiertes Literal:
$\exists! p_1, p_2, p_3 \in prp_V(map(r)) :$
$kv_P(p_1) = (kind, "literal") \wedge kv_P(p_2) = (literal, "$`lval`$")$
$\wedge kv_P(p_3) = (datatype, "dtm($`datatypeIRI`$)")$

(v) Falls $r \in LIT(R)$, $r =$ "`lval@langtag`" Literal mit Language-Tag:
$\exists! p_1, p_2, p_3, p_4 \in prp_V(map(r))$:
$kv_P(p_1) = (kind, "literal") \wedge kv_P(p_2) = (literal, "$`lval`$")$
$\wedge kv_P(p_3) = (datatype, "dtm($`rdf : langString`$)")$
$\wedge kv_P(p_4) = (language, "$`langtag`$")$
(Für Literal-Terme erhält der zugeordnete Knoten genau drei bzw.vier Propertys, welche die Art des RDF-Terms r, sowie Wert, Datentyp und, falls zutreffend, das Language-Tag erfassen.)

(nb-4) Die Funktionen vrt, lbl_E erfüllen folgende Nebenbedingung:
Sei $(r, p, o) \in R$. Es gilt:
$\exists!\, e \in E : vrt(e) = (v_1, v_2) \wedge v_1 = map(r) \wedge lbl_E(e) = p \wedge v_2 = map(o)$
(Jedem Tripel (r, p, o) entspricht genau eine Kante, deren Start- bzw. Endknoten der r bzw. o zugeordnete Knoten ist. Diese Kante ist mit dem Prädikat p gelabelt.)

Der zweite Transformationsansatz, den Hartig präsentiert, erzeugt im Ergebnisgraphen keine Knoten für die Literale. Stattdessen werden die Literal-Tripel zur Erzeugung von Propertys verwendet.

Während der resultierende LP-Graph der vorangehenden Transformation strukturell im Kern eng am gegebenen RDF-Graphen orientiert ist, entspricht das Ergebnis des sog. *Simple Property Graph*-Mappings damit eher den zu Beginn dieses Kapitels aus (Zhao et al. 2018) zitierten Bedeutungen der Strukturelemente von LP-Graphen, insbesondere der Propertys.

Die wesentlichen Ideen dieser Transformation sind:

- Für alle Subjekte und Objekte von Tripeln, welche IRI- oder Blank Node-Terme sind, wird ein zugehöriger Knoten im LP-Graphen erzeugt.
- Knoten, welche einem IRI-Term des RDF-Graphen zugeordnet sind, erhalten eine Property, welche diese IRI als Wert speichert. Für Blank Nodes ist ein analoges Vorgehen nicht notwendig, da ihre Identifikatoren nicht persistent sind.

- Für jedes Literal-Tripel wird in dem dem Tripel-Subjekt zugeordneten Knoten eine Property erzeugt, welche das Tripel-Prädikat als Key und das Tripel-Objekt (ein Literal!) als Wert erhält. D.h. die Attributierungen von Ressourcen im RDF-Graph führen nicht mehr zu Kanten, welche auf Literale verweisen, sondern werden nun in die diesen Ressourcen zugeordneten Knoten mittels Propertys integriert.

Tab. 7.2 und die Def. 7.2 erfassen Hartigs zweiten Ansatz im Detail. Abweichend von Hartigs Originalvorschlag betrachten wir jedoch auch hier lediglich normale RDF-Graphen als Input für die Transformation und keine RDF*-Graphen.

Tab. 7.2: Transformation der Komponenten eines RDF-Graphen R in den Labeled-Property-Graphen $LPG^{Hartig-Simple-PG}(R)$ gemäß Def. 7.2.

	Def. 7.2	Komponenten aus RDF-Graph R	Zugeordnete Komponenten in $LPG^{Hartig\text{-}Simple\text{-}PG}(R)$
(a)	(m-1) (m-5) (nb-5)(i)	$r \in IRI(R) \cap (SubjT(R) \cup ObjT(R))$	`map(r)` IRI [*dtm*(xsd:string)] = *r* … *(siehe c)*
(b)	(m-1) (m-5) (nb-5)(ii)	$r \in BN(R) \cap (SubjT(R) \cup ObjT(R))$	`map(r)` *(siehe c)*
(c)	(m-1) (m-5) (nb-5)(ii)	$r \in (IRI(R) \cup BN(R)) \cap (SubjT(R) \cup ObjT(R))$, $R_{LIT}(r) = \{t_1, \ldots, t_n\} \neq \emptyset$ mit $t_i = (r, d_i, l_i), 1 \leq i \leq n$ Für alle $t_j \in \{1, \ldots, n\}$, mit l_j einfaches Literal: $l_j =$ `value`$_j \in LIT(R)$, Für alle $t_k \in \{1, \ldots, n\}$, mit l_k typisiertes Literal: $l_k =$ `value`$_k$`^^datatypeIRI` $\in LIT(R)$, Für alle $t_m \in \{1, \ldots, n\}$, mit l_m Literal mit Language-Tag: $l_m =$ `value`$_m$`@lang` $\in LIT(R)$:	`map(r)` *(siehe a,b)* …, d_j[*dtm*(xsd:string)] = `value`$_j$, …, d_k[*dtm*(datatypeIRI)] = `value`$_k$, …, d_m[*dtm*(rdf:langString)] = `value`$_m$`@lang` …
(d)	(m-1) (nb-4)	$(r, p, o) \in R_{OBJ}(R)$	`map(r)` *(siehe a,b)* —**eid| p**→ `map(o)` *(siehe a,b)*
Ergebnistyp von $LPG^{Hartig\text{-}Simple\text{-}PG}(R)$: $\frac{(0,1)}{(+,0)}$			

Definition 7.2. (vgl. Hartig (2014), Def. 9, S. 14)

Sei R ein RDF-Graph.
Sei $dtm: \mathfrak{D}_{RDF} \to \mathfrak{D}$ eine Abbildung, welche Datentypen des RDF-Graphen (bzw. deren Bezeichner) in Datentypen der Property-Values (bzw. deren Bezeichner) abbildet.
Es gelte $dtm(\texttt{xsd:string}) = \mathcal{S}tr$ [7].

Es gelte, dass

(a) $PredT(R_{OBJ})$ und $PredT(R_{LIT})$ disjunkt sind
(Es gibt keinen Prädikat-Term, welcher sowohl eine Datatype- wie eine Objekt-Property bezeichnet.)

(b) alle Literal-Tripel in R mit selbem Prädikat, Literale mit selbem expliziten oder impliziten Typ[8]

[7] $\mathcal{S}tr$ bezeichnet den Datentyp aller Strings (siehe Def. 3.1).
[8] Mit implizitem Typ sind die Datentypen xsd:string und rdf:langString für einfache Literale bzw. Literale mit Language-Tag gemeint.

als Tripel-Objekte haben.
(Alle Literale, welche Werte derselben Datatype Property beschreiben, haben denselben Typ.[9])

$LPG^{Hartig-Simple-PG}(R) = (V, E, \mathcal{L}_V, \mathcal{L}_E, id_V, id_E, vrt, lbl_V, lbl_E, \mathcal{K}, \mathcal{V}, P, kv_P, prp_V, prp_E)$

bezeichnet den *Labeled-Property-Graphen*, welcher dem RDF-Graphen R gemäß Hartigs *Simple Property Graph*-Transformation zugeordnet wird.

Für diesen gilt ergänzend zu Def. 3.5:

(m-1) $|V| = |(IRI(R) \cup BN(R)) \cap (SubjT(R) \cup ObjT(R))|$
$map : (IRI(R) \cup BN(R)) \cup (SubjT(R) \cap ObjT(R)) \rightarrow V$ sei eine totale bijektive Funktion.
(Im zugeordneten LP-Graphen gibt es für alle IRIs und Blank-Nodes, welche Tripel-Subjekt oder Tripel-Objekt sind, genau einen Knoten. Dies schließt Knoten für Literale und IRIs, welche ausschließlich als Tripel-Prädikate vorkommen, aus.)

(m-2) $|E| = |R_{OBJ}|\ [= |R \setminus R_{LIT}|\,]$
(Die Anzahl der Kanten entspricht der Anzahl der Tripel in R ohne Literal-Tripel.)

(m-3) $\mathcal{L}_V = \emptyset$
(Es gibt keine Knotenlabel.)

(m-4) $\mathcal{L}_E = PredT(R_{OBJ})\ [= PredT(R \setminus R_{LIT})\,]$
(Alle Tripel-Prädikate Object Propertys bezeichnen, können Kantenlabel sein.)

(m-5) $\mathcal{K} = \{k \mid \exists\, t \in R_{LIT} : k = pred(t)\} \cup \{IRI\}$[a]
Sei $k \in \mathcal{K}_V$ und $t \in R_{LIT} : k = pred(t)$, dann gilt:

$$dt(k) = \begin{cases} \mathcal{S}tr & \text{falls } obj(t) \text{ einfaches Literal} \\ dtm(\texttt{datatypeIRI}) & \text{falls } obj(t) \text{ enthält } \texttt{\^\^datatypeIRI} \\ dtm(\texttt{rdf:langString}) & \text{falls } obj(t) \text{ Literal mit Language} - \text{Tag} \end{cases}$$

$$dt(IRI) = \mathcal{S}tr$$

(Alle Prädikate des Literal-Graphen von R (d.h. jene in R, die Datatype Propertys bezeichnen), sind Keys. Sind die zugehörigen Literale typisiert, dann erhält der Key die entsprechende Datentyp-IRI als Datentyp. Sind die zugehörigen Literale einfach oder mit Language-Tag versehen, so haben die Keys den Datentyp xsd:string bzw. rdf:langString.)

[a] Der Key-Bezeichner IRI ist ungleich aller vorkommenden Bezeichner der Datatype Propertys in R gewählt.

(f-8) $prp_E : prp_E(e) = \emptyset,\ p \in P$
(Kanten haben keine Propertys.)

(nb-4) Die Funktionen vrt und lbl_E erfüllen folgende Nebenbedingung:
Für alle Tripel $(r, p, o) \in R_{OBJ}(r)$:
$\exists!\, e \in E :\ vrt(e) = (v_1, v_2) \wedge v_1 = map(r) \wedge lbl_E(e) = \{p\} \wedge v_2 = map(o)$
(Jedem Objekt-Tripel (r, p, o) von r entspricht genau eine Kante mit Label p, deren Start- bzw. Endknoten der r bzw. o zugeordnete Knoten ist.)

[9] Diese Annahme stellt sicher, dass ein erzeugter Key einen eindeutigen Datentyp hat.

(nb-5) Die Funktionen kv_P und prp_V erfüllen folgende Nebenbedingung:
Sei $r \in (IRI(R) \cup BN(R)) \cup (SubjT(R) \cup ObjT(R))$. Es gilt:

(i) Falls $r \in IRI(R)$: $\exists! p \in prp_V(map(r)) : kv_P(p) = (IRI, "r")$
(Ein Knoten, der eine IRI r repräsentiert, erhält genau eine Property mit Key IRI und Value r.)

(ii) Sei $(r, d, l) \in R_{LIT}(r)$. Dann gilt:
$\exists! p \in prp_V(map(r)) : kv_P(p) = (d, "val")$ mit
$$val = \begin{cases} \texttt{value} & \text{falls } l = \texttt{value\^\^datatypeIRI typisiertesLiteral} \\ l & \text{sonst} \end{cases}$$ [a]
(Jedes Literal-Tripel einer Ressource r wird auf genau ein Key-Value-Paar einer Property des zugeordneten Knotens abgebildet. Bei typisierten Literalen ist der Datentyp nicht Teil des übernommenen Values, er wird dem Key zugeordnet (s.o.). Language-Tags werden dagegen übernommen, da sonst die Sprachkennzeichnung verloren gehen würde.)

[a] Beachte: Das Language-Tag ist Teil des Property-Wertes.

Abb. 7.2, S. 60, zeigt das Ergebnis der Transformation unseres RDF-Graphen RDF^{Pets}. Diese Visualisierung macht deutlich, dass die Knoten alle ungelabelt sind. Betrachtet wir exemplarisch den Knoten v26. Man erkennt an der vorhandenen Property mit Key *IRI* sowie deren Value, dass und welchem IRI-RDF-Term dieser Knoten zugeordnet wurde. Alle IRIs der Datatype Propertys des Literal-Graphen von exd: Bliss sind zu Keys geworden mit den entsprechenden Literalen als Values. Das Language Tag des Literals Bliss@en kann - anders als in der vorangehenden Transformation - nicht einem gesonderten Key *language* zugeordnet werden, weil dann die Verbindung zum Key exs:name verloren gehen würde. Am Knoten v24 erkennt man die Behandlung eines Blank Node-RDF-Terms. Die im RDF-Graphen vorhandene ID wird nicht übernommen.

7.2 Transformationen in Labeled-Property-Graphen basierend auf Thakkar

Thakkar (2021)[10] beschäftigt sich eingehend mit der Transformation von RDF-Graphen in das Labeled-Property-Graph-Modell. Als Zwischenrepräsentation für seine Transformationen verwendet er eine Formalisierung des gegebenen RDF-Graphen als Labeled-Graph. Diese Formalisierung haben wir bereits in Abschnitt 6.3 mit einigen von uns vorgenommenen Modifikationen betrachtet.

Insgesamt macht Thakkar drei Vorschläge für eine Formalisierung als Labeled-Property-Graph, welche er *Simple Database Mapping*, *Generic Database Mapping* und *Complete Database Mapping* nennt (siehe Abschn. 4.3.1, 4.3.2 und 4.3.1 in (Thakkar 2021)). Die Besonderheit seines Vorgehens ist, dass der Autor unterschiedliche Einbeziehungen eines sog. *Property Graph Schemas* in das Mapping betrachtet. Dieses Schema ist selbst als Labeled-Property-Graph repräsentiert. Oben genannte drei Varianten unterscheiden sich dahingehend, dass in der ersten Variante kein solches Schema mit einbezogen wird, im zweiten Fall ein generisches, d.h. vom gegebenen RDF-Graph unabhängiges Schema, und im dritten Ansatz ein Schema, welches aus einem RDFS-Schema durch ein sog. *Schema Mapping* erzeugt wird. In allen drei Varianten geht er davon aus, dass ein RDF-Graph gegeben ist, welcher die Datenebene (Instanzenebene) der betrachteten Domäne beschreibt. Dieser Graph wird mittels eines sog. *Instance Mappings* transformiert. Für das *Complete Database Mapping* trifft er die ergänzende Annahme, dass zusätzlich ein RDF-Graph vorhanden ist, welcher ein RDFS-Schema enthält, welches die im Daten-RDF-Graph verwendete Terminologie spezifiziert.

[10] siehe alternativ auch Angles et al. (2020b)

Wie bereits eingangs in Kap. 5 erwähnt, betrachten wir in diesem Bericht keine gesonderten Transformationen von in RDF formulierten Schema-Graphen, egal auf welchem Schemavokabular basierend. Betrachtet man Thakkars *Instance Mappings*, so ist jedoch nicht erkennbar, weshalb sie nicht auch auf schemabeschreibende RDF-Graphen angewendet werden können. Denn solche RDF-Graphen sind letztlich Instanzgraphen bzgl. der generischen epistemologischen Beschreibungsebene.

Im Weiteren konzentrieren wir uns auf die beiden *Instance Mappings* $\mathcal{IM}_2$ und $\mathcal{IM}_3$, welche Thakkar für sein *Generic Database Mapping* bzw. sein *Complete Database Mapping* erarbeitet hat.

Thakkars Grundidee seiner $\mathcal{IM}_2$-Transformation eines RDF-Graphen R lässt sich folgendermaßen zusammenfassen:

- Alle Tripel-Subjekte- oder -Objekte werden durch genau einen Knoten repräsentiert.
- Abhängig von der Art des RDF-Terms - IRI, Blank Node oder Literal -, den sie repräsentieren, erhalten Knoten
  - ein spezifisches Label, sowie
  - eine spezifische Property, welche den entsprechenden RDF-Term als String-Wert erhält.
- Die Topologie des gegebenen RDF Graphen soll weitestgehend erhalten bleiben. Lediglich die Typisierungs-Tripel werden nicht durch Kanten repräsentiert. Als Konsequenz werden z.B. die Schema- und die Datenebene, welche im RDF-Graph durch rdf:type-Tripel verknüpft sind, im entstehenden LP-Graphen topologisch entkoppelt.
- Für jedes Tripel $t = (s, p, o)$ aus R, welches kein Typisierungs-Tripel ist, wird eine gerichtete Kante zwischen den Knoten erzeugt, welche die RDF-Terme s und o repräsentieren. Diese wird in Abhängigkeit von der Art der Property p - Object oder Datatype Property - gelabelt.
- Thakkar trifft die Annahme, dass Nicht-Literal-Ressourcen r im zugrunde liegenden RDF-Graphen max. ein Typisierungs-Tripel besitzen. Liegt kein Typisierungtripel für r vor, so wird defaultmäßig die Zugehörigkeit zur Klasse `rdfs:Resource` angenommen.[11]. Hierdurch kann jedem Nicht-Literal-Knoten genau eine Property zugeordnet werden, welche die Typ-IRI als Wert erhält.

Wir formalisieren diese Transformation in Tab.7.3 sowie Def. 7.3 mit folgenden Anpassungen:

- Thakkar verwendet in seinem Paper die Label *Resource*, *BlankNode* und *Literal*. Wir ersetzen das Label *Resource* durch *IRI*, da Blank Nodes und Literale in RDF auch Ressourcen sind.
- Anstelle einer festgelegten Klasse, welche bei fehlender Typisierung einer Ressource als Default verwendet wird, sollte diese Defaultklasse für das Mapping konfigurierbar sein. Statt `rdfs:Resource` ist in anderen Kontexten vielleicht `owl:Thing` sinnvoller. Repräsentiert der gegebene RDF-Graph R nur die Datenebene einer Domäne, so könnte auch eine allgemeinste domänenspezifische Klasse als Default sinnvoll sein, der alle Objekte dieser Domäne umfasst.
- Die Annahme, dass es für Ressourcen max. ein Typisierungs-Tripel gibt, geben wir auf. Ressourcen r dürfen mehrere explizite Typisierungen haben, d.h. der Typisierungs-RDF-Graph von r mehr als ein Tripel enthalten. Diese Erweiterung trägt der Tatsache Rechnung, dass im RDF-Graph Multihierarchien repräsentiert sein können und eine Ressource verschiedenen dieser Hierarchien zugeordnet wird. Zudem können auch durch Inferenz die impliziten Typzugehörigkeiten materialisiert worden sein und so Ressourcen Subjekt mehrerer Typisierungs-Tripel sein.
- Thakkar verwendet als Ausgangspunkt seines sog. *Instance Mapping* $\mathcal{IM}_2$ einen Labeled-

[11] Siehe auch Thakkars Ansatz für einen L-Graphen, den wir Def. 6.3 zugrunde gelegt haben.

Tab. 7.3: Transformation der Komponenten eines RDF-Graphen R in den Labeled-Property-Graphen $LPG^{mod-Thakkar-IM2}(R)$ gemäß Def. 7.3

	Def. 7.3	Komponenten aus RDF-Graph R $IRI_{RootClass} \in \mathcal{IRI}$	Zugeordnete Komponenten in $LPG^{mod\text{-}Thakkar\text{-}IM2}(R)$
(a)	(m-1) (f-4) (m-5) (nb-3)(i)	$r \in IRI(R) \cap (SubjT(R) \cup ObjT(R \backslash R_{TYPES}))$	**IRI** map(r) iri [dtm(xsd:string)] = "r" … (siehe c1,c2)
(b)	(m-1) (f-4) (m-5) (nb-3)(ii)	$r \in BN(R) \cap (SubjT(R) \cup ObjT(R \backslash R_{TYPES}))$	**Blank Node** map(r) id [dtm(xsd:string)] = "r" … (siehe b1,b2)
(c1)	(m-1) (f-4)	$r \in (IRI(R) \cup BN(R)) \cap (SubjT(R) \cup ObjT(R \backslash R_{TYPES}))$ $R_{TYPE}(r) = \{t_1, \dots, t_n\} \neq \emptyset$ mit $t_i = (r, \texttt{rdf:type}, c_i), 1 \leq i \leq n$	**(siehe a1,a2)** map(r) type[dtm(xsd:string)] = "c_1" … type[dtm(xsd:string)] = "c_n" … (siehe a,b)
(c2)	(m-1) (f-4) (m-5)	$r \in (IRI(R) \cup BN(R)) \cap (SubjT(R) \cup ObjT(R \backslash R_{TYPES}))$ $R_{TYPE}(r) = \emptyset$	**(siehe a1,a2)** map(r) type[dtm(xsd:string)] = "$IRI_{RootClass}$" … (siehe a,b)
(d)	(m-1) (f-4) (m-5) (nb-3)(iv)	$r \in LIT(R)$ r einfaches Literal: r typisiertes Literal: r = lval^^datatypeIRI ∈ $LIT(R)$, r Literal mit Language-Tag: r = lval@lang ∈ $LIT(R)$:	**Literal** map(r) value [dtm(xsd:string)] = "r" type [dtm(xsd:string)] = "xsd:string" ODER value [dtm(xsd:string)] = "lval" type [dtm(xsd:string)] = "datatypeIRI" ODER value [dtm(xsd:string)] = "r" type [dtm(xsd:string)] = "rdf:langString"
(e1)	(m-1) (nb-4)	$(r,p,o) \in (R_{OBJ} \backslash R_{TYPE})$	**(siehe a1,a2)** map(r) (siehe a,b,c1,c2) — eid| **Object Property** → **(siehe a1,a2)** map(o) (siehe a,b,c1,c2)
(e2)	(m-1) (nb-4)	$(r,p,o) \in (R_{LIT})$	**(siehe a1,a2)** map(r) (siehe a,b,c1,c2) — eid| **Datatype Property** → **(siehe c)** map(r) / (siehe d)
Ergebnistyp von $LPG^{mod\text{-}Thakkar\text{-}IM2}(R)$: $\frac{(1,1)}{(+,1)}$ [gem. orig. Thakkar(2021) $\frac{(1,1)}{(+u,1)}$]			

Graphen, welcher aus R als Zwischenrepräsentation erzeugt wird[12]. Wir beschreiben die Transformation dagegen *direkt ausgehend von einem RDF-Graphen* R und integrieren hierbei jene Aspekte von Thakkars Zwischenrepräsentation als L-Graph, welche für den Ergebnis-LP-Graphen relevant sind. Zudem integrieren wir unsere in Def. 6.3 gemachten Modifikationen dieses L-Graphen.

Definition 7.3. (vgl. Thakkar (2021), Def. 12, S. 79)

Sei R ein RDF-Graph und $IRI_{RootClass}$ eine IRI.
Sei $dtm: \mathfrak{D}_{RDF} \to \mathfrak{D}$ eine Abbildung, welche Datentypen des RDF-Graphen (bzw. deren Bezeichner) in Datentypen der Property-Values abbildet (bzw. deren Bezeichner).
Es gelte $dtm(\texttt{xsd:string}) = \mathcal{S}tr$ [13].

Es gelte, dass

(a) $PredT(R_{OBJ})$ und $PredT(R_{LIT})$ disjunkt sind
(Es gibt keinen Prädikat-Term, welcher sowohl eine Datatype- wie eine Objekt-Property bezeichnet.)

(b) alle Literal-Tripel in R mit selbem Prädikat, Literale mit selbem expliziten oder impliziten Typ[14] als Tripel-Objekte haben.
(Alle Literale, welche Werte derselben Datatype Property beschreiben, haben denselben Typ.[15])

$LPG^{mod-Thakkar-IM2}(R) = (V, E, \mathcal{L}_V, \mathcal{L}_E, id_V, id_E, vrt, lbl_V, lbl_E, \mathcal{K}, \mathcal{V}, P, kv_P, prp_V, prp_E)$

bezeichnet den durch Transformation des RDF-Graphen R auf Basis von Thakkkars $\mathcal{IM}_2$ und unseren Modifikationen zugeordneten *Labeled-Property-Graphen*.

Für diesen gilt ergänzend zu Def. 3.5:

(m-1) $|V| = |SubjT(R) \cup ObjT(R \backslash R_{TYPES})|$
$map: SubjT(R) \cup ObjT(R \backslash R_{TYPES}) \to V$ sei eine totale bijektive Funktion.
(Im zugeordneten LP-Graphen gibt es für alle Tripel-Subjekte genau einen Knoten, sowie für alle Objekte von Nicht-Typisierungs-Tripeln.)

(m-2) $|E| = |R \setminus R_{Type}|$
(Die Anzahl der Kanten entspricht der Anzahl der Tripel in R ohne Typisierungs-Tripel.)

(m-3) $\mathcal{L}_V = \{IRI, BlankNode, Literal\}$
(Für jede Art von RDF-Termen gibt es je ein Knoten-Label.)

(m-4) $\mathcal{L}_E = \{ObjectProperty, DatatypeProperty\}$
(Für die beiden Property-Arten gibt es je ein Kanten-Label.)

(m-5) $\mathcal{K} = \{type, iri, id, value\}$
Es gilt:
$$dt(type) = dt(iri) = dt(id) = dt(value) = dtm(\texttt{xsd:string}) = \mathcal{S}tr$$
(Insgesamt werden nur vier Keys verwendet, welche alle den Datentyp String haben.[a])

[a] Diese Festlegung haben wir aus dem in shortcitePThakkar:2021a, Def. 11, für den Ergebnisgraphen festgelegten *Generic Property Graph Schema* abgeleitet.

[12] Eine Modifikation von Thakkars L-Graphen, welchen er *RDF Graph* nennt, haben wir in Def. 6.3 formalisiert. Siehe auch die zu dieser Def. vorangehenden Bemerkungen.
[13] $\mathcal{S}tr$ bezeichnet den Datentyp aller Strings (siehe Def. 3.1).
[14] Mit implizitem Typ sind die Datentypen xsd:string und rdf:langString für einfache Literale bzw. Literale mit Language-Tag gemeint.
[15] Diese Annahme stellt sicher, dass ein erzeugter Key einen eindeutigen Datentyp hat.

(f-4) lbl_V: Sei $r \in SubjT(R) \cup ObjT(R \backslash R_{TYPES})$.

$$lbl_V(map(r)) = \begin{cases} \{IRI\} & \text{falls } r \in IRI(R) \\ \{BlankNode\} & \text{falls } r \in BN(R) \\ \{Literal\} & \text{falls } r \in LIT(R) \end{cases}$$

(Jeder Knoten erhält das Label, welches die Art des RDF-Terms kennzeichnet, den er repräsentiert.)

(nb-3) Die Funktionen kv_P und prp_V erfüllen folgende Nebenbedingung:
Sei $r \in SubjT(R) \cup ObjT(R \backslash R_{TYPES})$. Es gilt:

(i) Sei $r \in IRI(R)$.
$\exists! p \in prp_V(map(r)) : kv_P(p) = (iri, "r")$
(Für IRI-Terme r erhält der zugeordnete Knoten genau eine Property mit Key iri und Wert r.)

(ii) Sei $r \in BN(R)$.
$\exists! p \in prp_V(map(r)) : kv_P(p) = (id, "r")$
(Für Blank Node-Terme r erhält der zugeordnete Knoten genau eine Property mit Key id und Wert r.)

(iii) Sei $r \in IRI(R) \cup BN(R)$.
Für alle $o \in Obj(R_{TYPE}(r))$ gilt:
$\exists! p \in prp_V(map(r)) : \ kv_P(p) = (type, "o")$.
Falls $Obj(R_{TYPE}(r)) = \emptyset$
$\exists! p \in prp_V(map(r)) : \ kv_P(p) = (type, "IRI_{RootClass}")$.
(Für IRI- bzw. Blank Node-Terme r erhält der zugeordnete Knoten für jeden Typ von r genau eine Property mit Key $type$ und Wert r. Ist der Term nicht typisiert wird der gegebene Default-Typ zugeordnet.)

(iv) Sei $r \in LIT(R)$.
$\exists! p_1, p_2 \in prp_V(map(r)) : kv_P(p_1) = (value, "v") \wedge kv_P(p_2) = (type, "t")$ mit:
$v = r \wedge t = \texttt{xsd:String}$, falls r einfaches Literal,
$v = \texttt{lval} \wedge t = \texttt{datatypeIRI}$,
falls $r = \texttt{lval\^\^datatypeIRI}$ typisiertes Literal,
$v = r \wedge t = \texttt{rdf:langString}$, falls r Literal mit Language − Tag
(Für Literal-Terme erhält der zugeordnete Knoten genau zwei Propertys mit Key $value$ bzw. $type$. Die zugeordneten Werte sind abhängig von der Art des Literals. Zu beachten ist, dass Language-Tags in den Wert des Keys $value$ übernommen werden, da sonst die Sprachkennzeichnung verloren gehen würde.)

(nb-4) Die Funktionen vrt, lbl_E, kv_P und prp_E erfüllen folgende Nebenbedingung:
Sei $(r, p, o) \in R \backslash R_{TYPES}$. Es gilt:
$\exists!\, e \in E$:

$$vrt(e) = (v_1, v_2) \ \wedge \ v_1 = map(r) \ \wedge \ v_2 = map(o)$$
$$\wedge\, lbl_E(e) = \begin{cases} \{ObjectProperty\} & \text{falls } (r,p,o) \text{ Objekt} - \text{Tripel} \\ \{DatatypeProperty\} & \text{falls } (r,p,o) \text{ Literal} - \text{Tripel} \end{cases}$$
$$\wedge \exists! p \in prp_E(map(e)) : kv_P(p) = (type, o)$$

(Jedem Nicht-Typisierungs-Tripel (r, p, o) entspricht genau eine Kante, deren Start- bzw. Endknoten der r bzw. o zugeordnete Knoten ist. Diese Kante ist gemäß Art der Property p gelabelt und besitzt genau eine Property mit Key $type$ und Wert o.)

Die $IM2$-Transformation von Thakkar kann als Abwandlung von Hartigs *RDF-like-PG*-Transformation gesehen werden. Folgende wesentliche Unterschiede sind hervorzuheben und anhand der Abb. 7.3, S. 61, im Vergleich zu Abb. 7.1, S. 59, nachvollziehbar:

- Statt einer spezifischen Property nutzt Thakkar ein Knotenlabel, um die Art des jeweiligen RDF-Terms zu repräsentieren, welcher einem Knoten zugrunde liegt.
  So erhalten die Knoten v21, v24 und v33 die Label *IRI*, *BlankNode* bzw. *Literal*.
- Kanten werden ausschließlich Objekt- und Literal-Tripeln zugeordnet. Liegt einer Kante ein Objekt-Tripel zugrunde, dann wird sie mit dem Label *ObjectProperty* versehen, da das Tripel-Prädikat eine Object Property bezeichnet. Analog erhalten Kanten, welche aus Literal-Tripeln hervorgehen, das Label *DatatypeProperty*.
  Entsprechend erhalten die Kanten e40 und e24, welche auf den Tripeln (exd:Kitty, exs:name "Kitty") bzw. (exd:John,exs: likes ,exd:Thing99) basieren, die Label *Datatype Property* bzw. *Object Property*.
- Die Typisierungs-Tripel verwendet Thakkar, um Knoten des LP-Graphen mittels einer spezifischen Property mit Key *type* zu typisieren.
  So führt das Tripel (exd: Bliss , rdf:type, exs:Dog) zur Property (type,"exs:Dog") im Knoten v26.
- Da den Typisierungs-Tripeln keine Kanten zugeordnet werden, sind die Beschreibungsebenen im Ergebnisgraphen voneinander topologisch getrennt, wie auch schon in der auf ihm basierenden L-Graph-Transformation in Abschn. 6.3 zu sehen.
- Auch Kanten erhalten eine Property, welche die Zugehörigkeit zu der Relation erfasst, welche durch das Prädikat des zugrunde liegenden Tripels ausgedrückt ist.
  So erhalten bereits oben genannten Kanten e40 und e24 die Propertys (type,"exs:name") bzw. (type, exs: likes).

In einem weiteren Transformationsvorschlag, *Instance Mapping* $\mathcal{IM}_3$ genannt, verfolgt Thakkar folgende zwei Kernideen:

- Die Typen werden zu Labeln.
- Die durch die Literalgraphen $R_{LIT}(r)$ für Ressourcen r beschriebenen Attributierungen werden dem Knoten für r als Propertys zugeordnet.

Beschreibt ein RDF-Graph die Datenebene einer Domäne, dann ist als Konsequenz dieser Vorgehensweise die topologische Struktur des entstehenden LP-Graphen im Kern auf die Vernetzung der Domänenobjekte reduziert[16].

Während der resultierende Labeled Property-Graph der vorangehenden Transformation $\mathcal{IM}_2$ strukturell im Kern eng am gegebenen RDF-Graphen orientiert ist (siehe auch obige Anmerkungen bzgl. $\mathcal{IM}_2$ und Hartigs Transformation in einen *RDF-like Property Graph*), berücksichtigt das $\mathcal{IM}_3$-Mapping die zu Beginn dieses Kapitels aus (Zhao et al. 2018) zitierten Bedeutungen der Strukturelemente von Labeled-Property-Graphen. Damit wiederum sind Parallelen zwischen dem $\mathcal{IM}_3$-Mapping und Hartigs *Simple Property Graph*-Transformation gegeben.

In Übereinstimmung mit Hartigs *Simple Property Graph*-Transformation gilt für Thakkars $\mathcal{IM}_3$-Mapping:

- Alle Tripel-Subjekte- oder -Objekte, die keine Literale sind, werden durch genau einen Knoten repräsentiert.
- In den IRI-Termen zugeordneten Knoten wird die IRI als Value einer spezifischen Property erfasst.

[16] Da Blank Nodes i.d.R. Instanzen mehrstelliger Relationen darstellen, müssen ergänzend auch diese als eigenständige Knoten erhalten und vernetzt bleiben.

Tab. 7.4: Transformation der Komponenten eines RDF-Graphen R in den Labeled-Property-Graphen $LPG^{mod-Thakkar-IM3}(R)$ gemäß Def. 7.4.

	Def. 7.4	Komponenten aus RDF-Graph R $IRI_{RootClass} \in \mathcal{IRI}$	Zugeordnete Komponenten in $LPG^{mod\text{-}Thakkar\text{-}IM3}(R)$
(a)	(m-1) (m-5) (nb-5)(i)	$r \in IRI(R) \cap (SubjT(R) \cup ObjT(R \backslash R_{TYPES}))$	**siehe (b1),(b2)** `map(r)` iri [*dtm*(xsd:string)] = *r* … *(siehe d)*
(b)	(m-1) (m-5) (nb-5)(ii)	$r \in BN(R) \cap (SubjT(R) \cup ObjT(R \backslash R_{TYPES}))$	**siehe (b1),(b2)** `map(r)` id [*dtm*(xsd:string)] = *r* … *(siehe d)*
(c1)	(m-1) (f-4)	$r \in (IRI(R) \cup BN(R)) \cap (SubjT(R) \cup ObjT(R \backslash R_{TYPES}))$ $R_{TYPE}(r) = \{t_1, \ldots, t_n\} \neq \emptyset$ mit $t_i = (r, \texttt{rdf:type}, c_i), 1 \leq i \leq n$	**c_1, …, c_n** `map(r)` *(siehe a,b,d)*
(c2)	(m-1) (f-4)	$r \in (IRI(R) \cup BN(R)) \cap (SubjT(R) \cup ObjT(R \backslash R_{TYPES}))$ $R_{TYPE}(r) = \emptyset$	**$IRI_{RootClass}$** `map(r)` *(siehe a,b,d)*
(d)	(m-1) (m-5) (nb-5)(iii)	$r \in (IRI(R) \cup BN(R)) \cap (SubjT(R) \cup ObjT(R \backslash R_{TYPES}))$, $R_{LIT}(r) = \{t_1, \ldots, t_n\} \neq \emptyset$ mit $t_i = (r, p_i, li), 1 \leq i \leq n$ Für alle $t_j \in \{1, \ldots, n\}$, mit l_j einfaches Literal: $l_j = \texttt{value}_j \in LIT(R)$, Für alle $t_k \in \{1, \ldots, n\}$, mit l_k typisiertes Literal: $l_k = \texttt{value}_k\texttt{\^\^datatypeIRI} \in LIT(R)$, Für alle $t_m \in \{1, \ldots, n\}$, mit l_m Literal mit Language-Tag: $l_m = \texttt{value}_m\texttt{@lang} \in LIT(R)$:	**siehe (b1), (b2)** `map(r)` *siehe (a1),(a2)* …, p_j [*dtm*(xsd:string)] = $\texttt{value}_j$, …, p_k [*dtm*(datatypeIRI)] = $\texttt{value}_k$, …, p_m [*dtm*(rdf:langString)] = $\texttt{value}_m\texttt{@lang}$ …

(e)	(m-1)
	 (nb-4)
		$(r, p, o) \in (R_{OBJ} \backslash R_{TYPE})$	**siehe (b1),(b2)**
			 `map(r)`
			 (siehe a,b,c)
			 `eid|p` →
			 siehe (b1),(b2)
			 `map(o)`
			 (siehe a,b,c)

Ergebnistyp von $LPG^{mod\text{-}Thakkar\text{-}IM3}(R)$: $\frac{(+,1)}{(+,0)}$			[gem. orig. Thakkar(2021) $\frac{(1,1)}{(+,0)}$]

- Alle Tripel des Objekt-Graphen von R, die keine Typisierungs-Tripel sind, werden zu Kanten, die mit dem jeweiligen Tripelprädikat, welches eine Object Property bezeichnet, gelabelt sind.
- Alle Literal-Tripel einer Ressource werden zu Propertys des Knotens, der diese Ressource repräsentiert.

Abweichend von Hartig gilt:

- Die ID von Blank Node-Termen wird in den zugeordneten Knoten als Value einer Property erfasst.
- Thakkar erlaubt lediglich maximal ein Typisierungs-Tripel je Ressource im gegebenen RDF-Graphen. Ein vorhandenes Typisierungs-Tripel einer Ressource bestimmt das Label des zugeordneten Knotens. Bei fehlendem Typisierungs-Tripel wird das Label rdfs:Resource als als Standardlabel zugeordnet.

Gegenüber Thakkars Originalvorschlag, machen wir folgende Erweiterungen:

- Wie bei der vorangehenden Transformation erlauben wir den verwendeten Default-Typ durch Angabe einer IRI explizit zu spezifizieren.
- Thakkar verwendet als Ausgangspunkt seines *Instance Mapping* $\mathcal{IM}_3$ denselben aus dem gegebenen RDF-Graphen R erzeugten L-Graphen wie bei der Transformation $\mathcal{IM}_2$ [17]. Wir beschreiben hier wiederum die Transformation *direkt ausgehend von einem RDF-Graphen* R. Dabei integrieren wir jene Aspekte von Thakkars L-Graph-Zwischenrepräsentation, welche für den Ergebnis-LP-Graphen relevant sind, sowie unsere in Def. 6.3 gemachten Modifikationen, insbesondere die Erweiterung, dass Ressourcen Subjekt mehrerer Typisierungs-Tripel sein dürfen.

Obige Tab.7.4 und folgende Definition erfassen diese modifizierte Fassung von Thakkars *Instance Mapping* $\mathcal{IM}_3$.

Definition 7.4. (vgl. Thakkar (2021), Def. 15, S. 85)

Sei R ein RDF-Graph und $IRI_{RootClass}$ eine IRI.
Sei $dtm: \mathfrak{D}_{RDF} \rightarrow \mathfrak{D}$ eine Abbildung, welche Datentypen des RDF-Graphen (bzw. deren Bezeichner) in Datentypen der Property-Values abbildet (bzw. deren Bezeichner).
Es gelte $dtm(\texttt{xsd:string}) = \mathcal{S}tr$ [18].

Es gelte, dass

(a) $PredT(R_{OBJ})$ und $PredT(R_{LIT})$ disjunkt sind
(Es gibt keinen Prädikat-Term, welcher sowohl eine Datatype- wie eine Objekt-Property bezeichnet.)

(b) alle Literal-Tripel in R mit selbem Prädikat, Literale mit selbem expliziten oder impliziten Typ[19] als Tripel-Objekte haben. *(Alle Literale, welche Werte derselben Datatyp-Property beschreiben, haben denselben Typ.[20])*

$$LPG^{mod-Thakkar-IM3}(R) = (V, E, \mathcal{L}_V, \mathcal{L}_E, id_V, id_E, vrt, lbl_V, lbl_E, \mathcal{K}, \mathcal{V}, P, kv_P, prp_V, prp_E)$$

bezeichnet den durch Transformation des RDF-Graphen R auf Basis von Thakkkars $\mathcal{IM}_3$ und unseren Modifikationen zugeordneten *Labeled-Property-Graphen*.

[17] Eine Modifikation von Thakkars L-Graphen, welchen er *RDF Graph* nennt, haben wir in Def. 6.3 formalisiert. Siehe auch die zu dieser Def. vorangehenden Bemerkungen.
[18] $\mathcal{S}tr$ bezeichnet den Datentyp aller Strings (siehe Def. 3.1).
[19] Mit implizitem Typ sind die Datentypen xsd:string und rdf:langString für einfache Literale bzw. Literale mit Language-Tag gemeint.
[20] Diese Annahme stellt sicher, dass ein erzeugter Key einen eindeutigen Datentyp hat.

Für diesen gilt ergänzend zu Def. 3.5:

(m-1) $|V| = |(IRI(R) \cup BN(R)) \cap (SubjT(R) \cup ObjT(R))|$
$map : (IRI(R) \cup BN(R)) \cup (SubjT(R) \cap ObjT(R)) \rightarrow V$ sei eine totale bijektive Funktion.
(Im zugeordneten LP-Graphen gibt es für alle IRIs und Blank-Nodes, welche Tripel-Subjekt oder Tripel-Objekt sind, genau einen Knoten. Dies schließt Knoten für Literale und IRIs, welche ausschließlich als Tripel-Prädikate vorkommen, aus.)

(m-2) $|E| = |R_{OBJ} \setminus R_{Type}|$ $[= |R \setminus (R_{LIT} \cup R_{TYPE})|]$
(Die Anzahl der Kanten entspricht der Anzahl der Tripel in R ohne Literal-Tripel und ohne Typisierungs-Tripel.)

(m-3) $\mathcal{L}_V = ObjT(R_{TYPE}) \cup \{IRI_{RootClass}\}$
(Alle RDF-Terme, welche zur Typisierung in Typisierungs-Tripeln von R verwendet werden, sowie die IRI der gegebenen Wurzelklasse können Knotenlabel sein.)

(m-4) $\mathcal{L}_E = PredT(R_{OBJ} \setminus R_{Type})$ $[= PredT(R \setminus (R_{LIT} \cup R_{TYPE}))]$
(Alle Tripel-Prädikate ungleich rdf:type, welche keine Datatype Propertys bezeichnen, können Kantenlabel sein.)

(m-5) $\mathcal{K} = \{k \mid \exists\, t \in R_{LIT} : k = pred(t)\} \cup \{iri, id\}$[a]
Sei $k \in \mathcal{K}_V$ und $t \in R_{LIT} : k = pred(t)$, dann gilt:

$$dt(k) = \begin{cases} dtm(\texttt{datatypeIRI}) & \text{falls } obj(t) \text{ enthält } \texttt{\^\^datatypeIRI} \\ \mathcal{S}tr & \text{falls } obj(t) \text{ einfaches Literal} \\ dtm(\texttt{rdf:langString}) & \text{falls } obj(t) \text{ Literal mit Language} - \text{Tag} \end{cases}$$

$dt(iri) = \mathcal{S}tr$
$dt(id) = \mathcal{S}tr$
(Alle Prädikate des Literal-Graphen von R (d.h. jene in R, die Datatype Propertys bezeichnen), sind Keys. Sind die zugehörigen Literale typisierte Strings, dann erhält der Key die entsprechende Datentyp-IRI als Datentyp. Sind die zugehörigen Literale einfache Strings oder Strings mit Language-Tags, so haben die Keys den Datentyp xsd:string bzw. rdf:langString.)

[a] iri und id sind ungleich aller vorkommenden Bezeichner der Datatype Propertys in R gewählt.

(f-4) Sei $r \in (IRI(R) \cup BN(R)) \cup (SubjT(R) \cup ObjT(R))$. Es gilt:

$$lbl_V(map(r)) = \begin{cases} ObjT(R_{TYPE}(r)) & \text{falls nicht leer} \\ \{IRI_{RootClass}\} & \text{sonst} \end{cases}$$

(Ein Knoten, der die Ressource r im RDF-Graphen repräsentiert, erhält alle jene Typen als Label, welche für r in R explizit angegeben sind. Ist r in R nicht explizit typisiert, dann wird die gegebene Wurzelklasse als Defaulttyp verwendet.)

(f-8) $prp_E : prp_E(e) = \emptyset,\ p \in P$
(Kanten haben keine Propertys.)

(nb-4) Die Funktionen vrt und lbl_E erfüllen folgende Nebenbedingung:
Für alle Tripel $(r, p, o) \in (R_{OBJ} \setminus R_{TYPE})$:
$\exists!\, e \in E : vrt(e) = (v_1, v_2) \wedge v_1 = map(r) \wedge lbl_E(e) = \{p\} \wedge v_2 = map(o)$
(Jedem Objekt-Tripel (r, p, o) von r entspricht genau eine Kante mit Label p, deren Start- bzw. Endknoten der r bzw. o zugeordnete Knoten ist.)

(nb-5) Die Funktionen kv_P und prp_V erfüllen folgende Nebenbedingung:
Sei $r \in (IRI(R) \cup BN(R)) \cup (SubjT(R) \cup ObjT(R))$. Es gilt:

(i) Falls $r \in IRI(R)$: $\exists! p \in prp_V(map(r)) : kv_P(p) = (iri, "r")$
(Ein Knoten, der eine IRI r repräsentiert, erhält genau eine Property mit Key iri und Value r.)

(ii) Falls $r \in BN(R)$: $\exists! p \in prp_V(map(r)) : kv_P(p) = (id, "r")$
(Ein Knoten, der einen Blank-Node r repräsentiert, erhält genau eine Property mit Key id und Value r.)

(iii) Sei $(r, d, l) \in R_{LIT}(r)$. Dann gilt:
$\exists! p \in prp_V(map(r)) : kv_P(p) = (d, "val")$ mit

$$val = \begin{cases} \texttt{value} & \text{falls } l = \texttt{value\^{}\^{}datatypeIRI typisiertesLiteral} \\ l & \text{sonst} \end{cases}$$ [a]

(Jedes Literal-Tripel einer Ressource r wird auf genau ein Key-Value-Paar einer Property des zugeordneten Knotens abgebildet. Bei typisierten Literalen ist der Datentyp nicht Teil des übernommenen Values, er wird dem Key zugeordnet (s.o.). Language-Tags werden dagegen übernommen, da sonst die Sprachkennzeichnung verloren gehen würde.)

[a] Beachte: Das Language-Tag ist Teil des Property-Wertes.

Thakkars *Instance Mapping* $\mathcal{IM}_1$, welches er als Teil seines *Simple Database Mapping* vorstellt, ist eine vereinfachte Variante des *Instance Mapping* $\mathcal{IM}_3$ dahingehend, dass

- auf alle Namespaces verzichtet wird und
- die Keys iri und id nicht erzeugt werden.

Durch den Verzicht auf die Namespaces können gleichbenamte Keys ggf. sogar im selben Knoten entstehen, die im RDF-Graphen verschiedene RDF-Propertys mit unterschiedlicher Bedeutung bezeichnet haben. Der Wegfall der Keys iri und id bewirkt den Verlust der im RDF-Graphen vorhandenen Identifikatoren. Eine Wiederherstellung des RDF-Graphen ist damit nicht möglich.

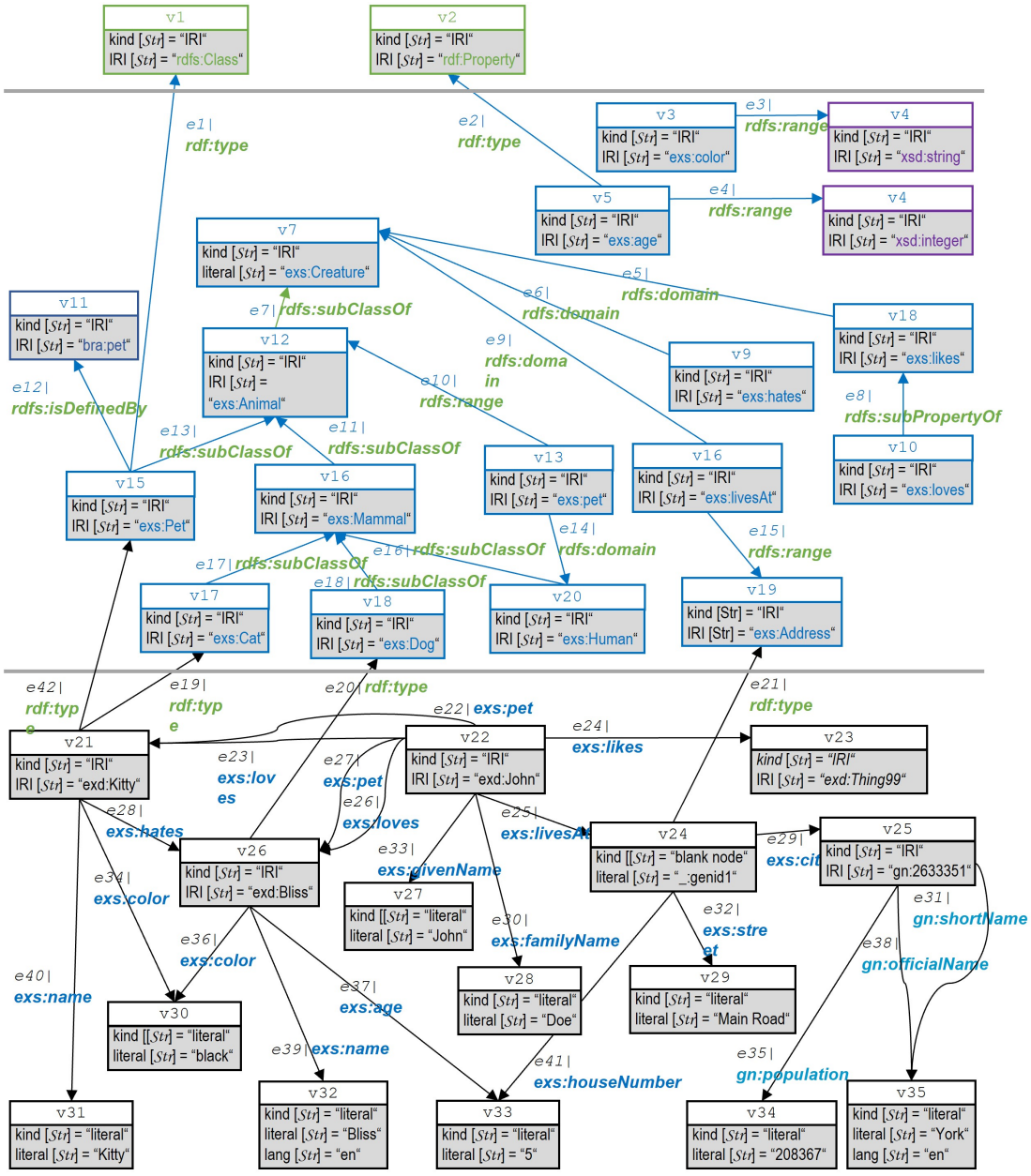

Abb. 7.1: Visualisierung des dem Beispiel-RDF-Graphen R^{Pets} aus Kap. 2 , S. 13 zugeordneten Labeled-Property-Graphen $LPG^{Hartig-RDF-like}(R^{Pets})$ gemäß Def. 7.1.

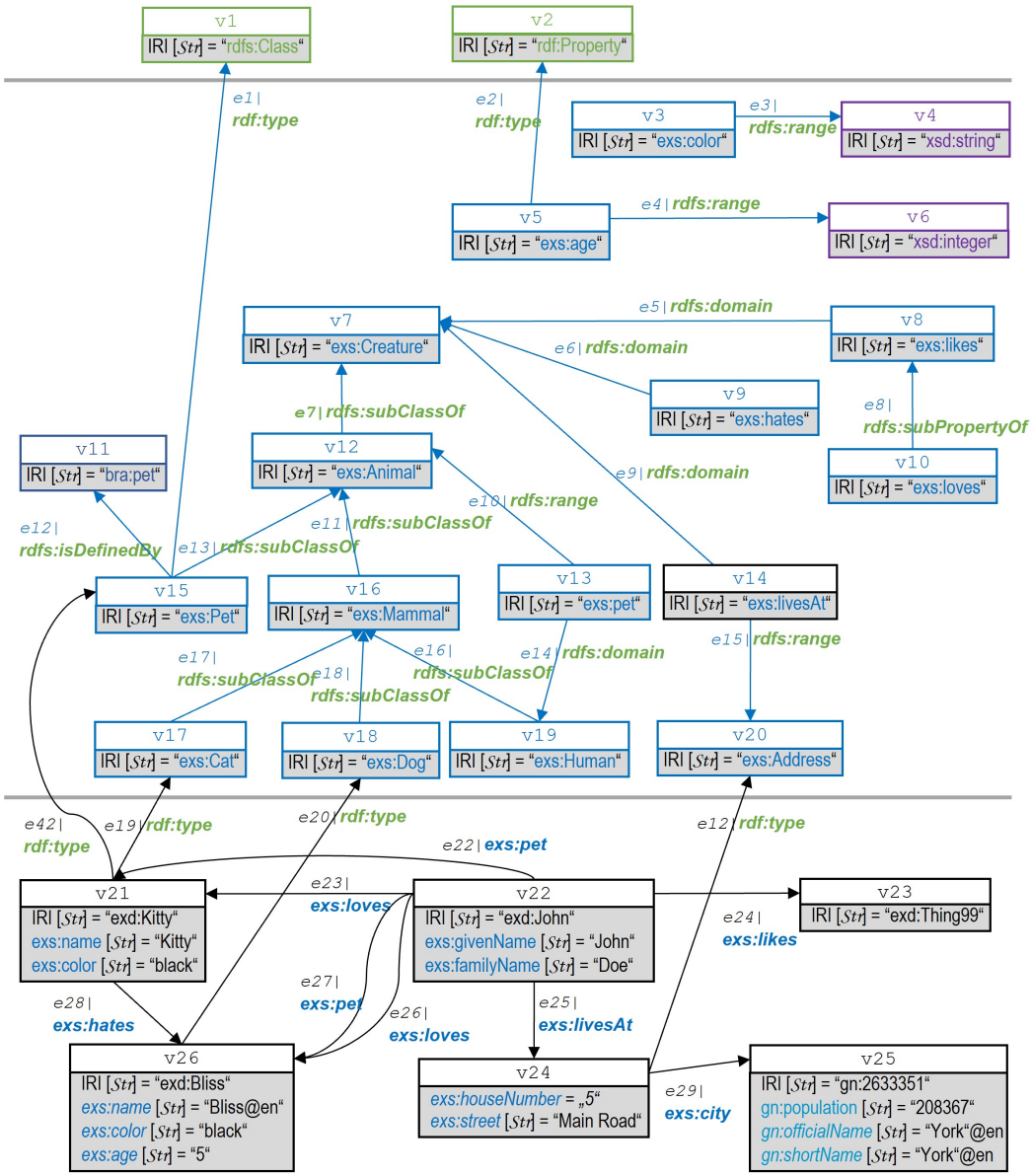

Abb. 7.2: Visualisierung des dem Beispiel-RDF-Graphen R^{Pets} aus Kap. 2 , S. 13 zugeordneten Labeled-Property-Graphen $LPG^{Hartig-Simple-PG}(R^{Pets})$ gemäß Def. 7.2.

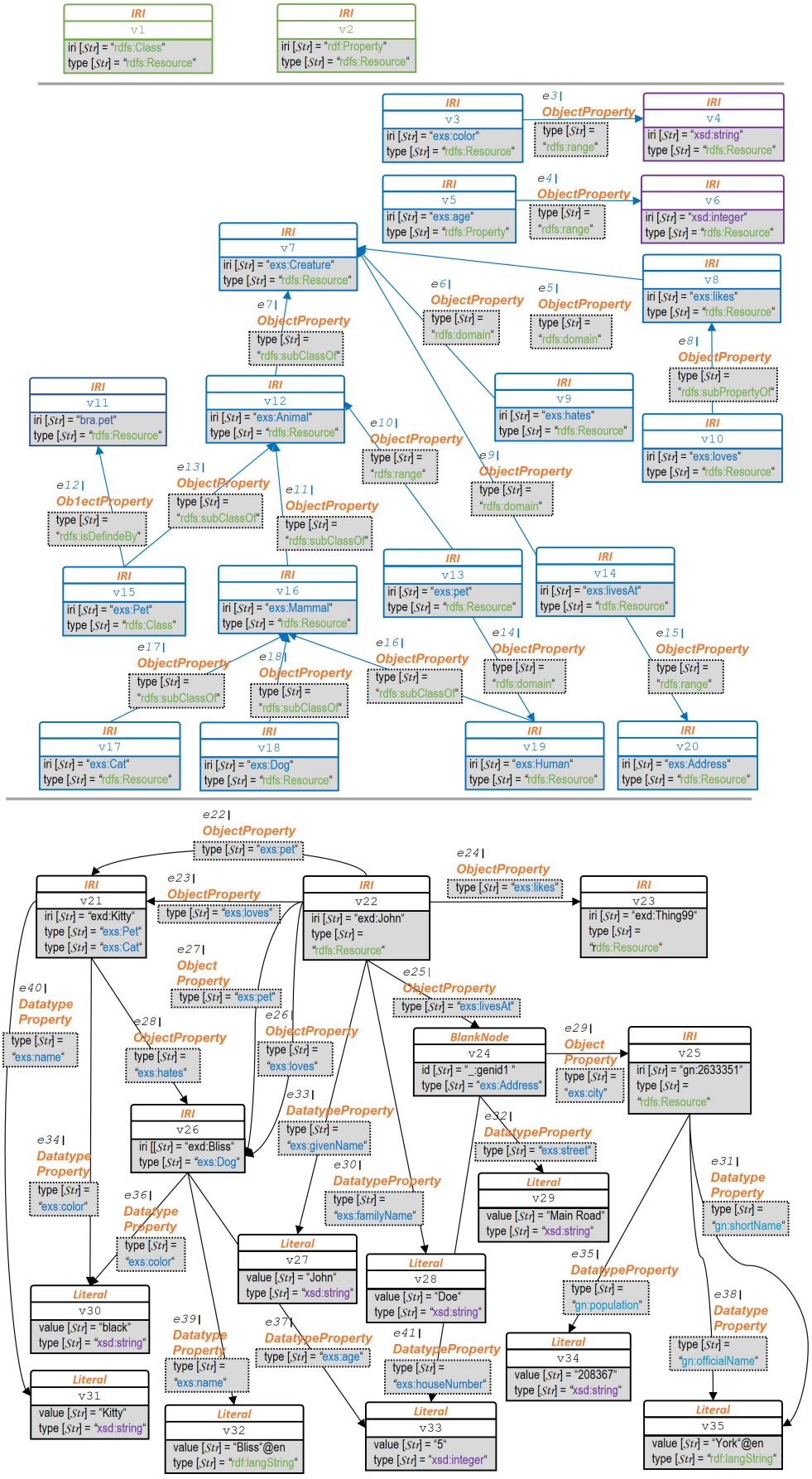

Abb. 7.3: Visualisierung des dem Beispiel-RDF-Graphen R^{Pets} aus Kap. 2 , S. 13 zugeordneten Labeled-Property-Graphen $LPG^{mod-Thakkar-IM2}(R^{Pets})$ gemäß Def. 7.3.

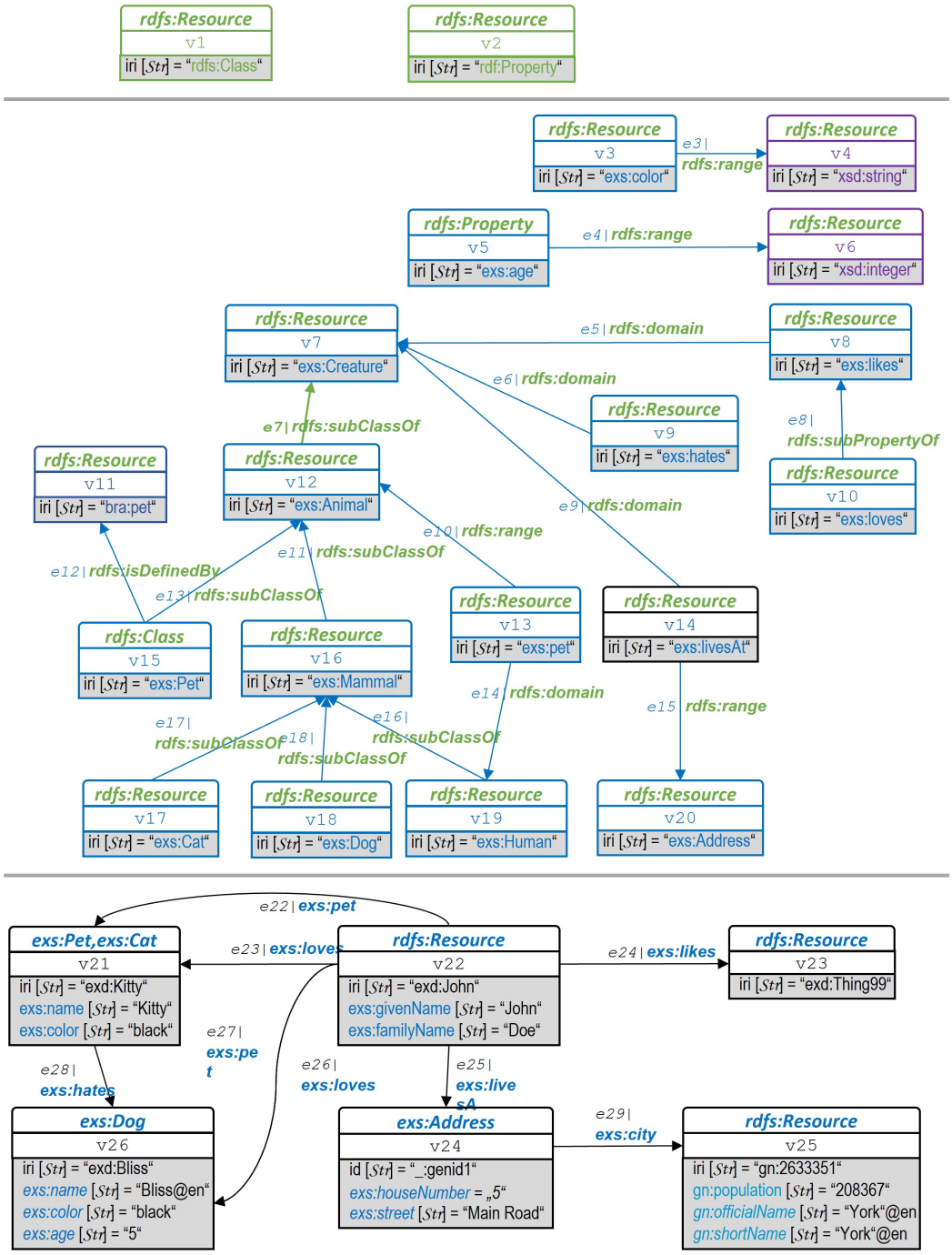

Abb. 7.4: Visualisierung des dem Beispiel-RDF-Graphen R^{Pets} aus Kap. 2 , S. 13 zugeordneten Labeled-Property-Graphen $LPG^{mod-Thakkar-IM3}(R^{Pets})$ gemäß Def. 7.4.

8 Zusammenfassende Bemerkungen

Die genaue Betrachtung von Graphdatenbanken offenbart, dass das zugrunde liegende Graphdatenmodell oft i. S. der mathematischen Graphentheorie nicht formalisiert vorliegt. Das RDF-Graph-Modell ist zwar standardisiert, jedoch als Menge sog. Tripel. Die Interpretation dieser Menge als formaler Graph wird lediglich mit Blick auf die Visualisierung des Datenmodells skizziert. Somit bleibt Spielraum für die formale Repräsentation. Labeled Property-Graphdatenbanken fehlt es an Standardisierung. Was genau unter einem Labeled-Property-Graphen verstanden wird, ist entsprechend unterschiedlich. In der Folge werden auch in Arbeiten, welche Aspekte der Interoperabilität der Graphmodelle betrachten, unterschiedliche Annahmen hinsichtlich der strukturellen Anforderungen an Labeled-Property-Graphen getroffen.

In diesem Bericht betrachten wir unterschiedliche Repräsentationen von RDF-Graphen als formale Graphen: Node-Pair-Labeled-Graphen, Labeled-Graphen und Labeled-Property-Graphen. Alle diese Graphen sind gerichtete, kanten- und/oder knotenmarkierte Multi-Graphen. Bei der einfachsten Variante, den NPL-Graphen, sind nur die Knoten First-Class-Strukturelemente, es gibt nur Label als Markierungen und dies auch nur an den Kanten. Bei L-Graphen sind zusätzlich die Kanten First-Class-Citizens und die Knoten können auch mit Labeln markiert werden. Schließlich wird bei LP-Graphen eine zweite Art von Markierungen ergänzt, die Propertys, welche ebenfalls First-Class-Strukturelemente sind. Mit ihnen können, ergänzend oder alternativ zu den Labeln, Knoten und Kanten markiert werden.

Bereits die durch das W3C in den Recommendations gewählten Visualisierungen von RDF-Graphen legen nahe, dass diese als formale Graphen repräsentierbar sind. Es ist somit nicht die Frage, ob RDF-Graphen als formale Graphen beschreibbar sind, die interessante Frage ist vielmehr, in welcher Weise sich RDF-Graphen durch welche unterschiedlichen Arten formaler Graphen repräsentieren lassen. Einige in der Literatur vorgestellten Möglichkeiten haben wir in diesem Bericht einheitlich aufbereitet und anhand eines übergreifenden Beispiel-RDF-Graphen illustriert.

Bonifati et al. (2018) und Tomaszuk et al. (2020) haben wertvolle Übersichten über die Vielfalt der relevanten formalen bzw. der in Systemen verwendeten Ausprägungen von Labeled Property-Graphen ergestellt. Unsere Definition von Labeled-Property-Graphen ist sehr allgemein gefasst, wobei wir uns auf Knoten, Kanten und Properties als First-Class-Strukturelemente beschränken.[1] Um eine differenzierte Betrachtung zu ermöglichen, haben wir in Kap. 3 eine Typ-Notation eingeführt, mit der unterschiedlichste Ausprägungen von Labeled Property-Graphen charakterisierbar sind. Hierauf basiert unsere Übersichtsdarstellung in Abschnitt 3.4, welche die in Bonifati et al. (2018), Tomaszuk et al. (2020) sowie weitere in der Literatur genannten Varianten von Labeled-Property-Graphen erfasst.

Da RDF-Graphen keine formalen Graphen sind, ist jede der vorgestellten Formalisierungen von RDF-Graphen das Ergebnis einer Transformation. Dieser Begriff wird in der Literatur naheliegenderweise verwendet, wenn der Aspekt der Interoperabilität der Graphmodelle im Vordergrund steht. Unser Bericht berücksichtigt exemplarisch vier Transformationen von RDF-Graphen in Labeled-Property-Graphen. In Abb. 8.1 haben wir abschließend die Typen der Ergebnisgraphen dieser Transformationen bzw. der ihnen zugrunde liegenden originalen Ansätze in die Übersicht aus Abschn. 3.4, S. 26,

[1] Für die Betrachtung von Graph Queries sind weitere komplexere Strukturelemente relevant, wie z.B. Pfade oder Subgraphen, was zu entsprechend komplexeren Graphmodellen führt. Siehe hierzu Bonifati et al. (2018).

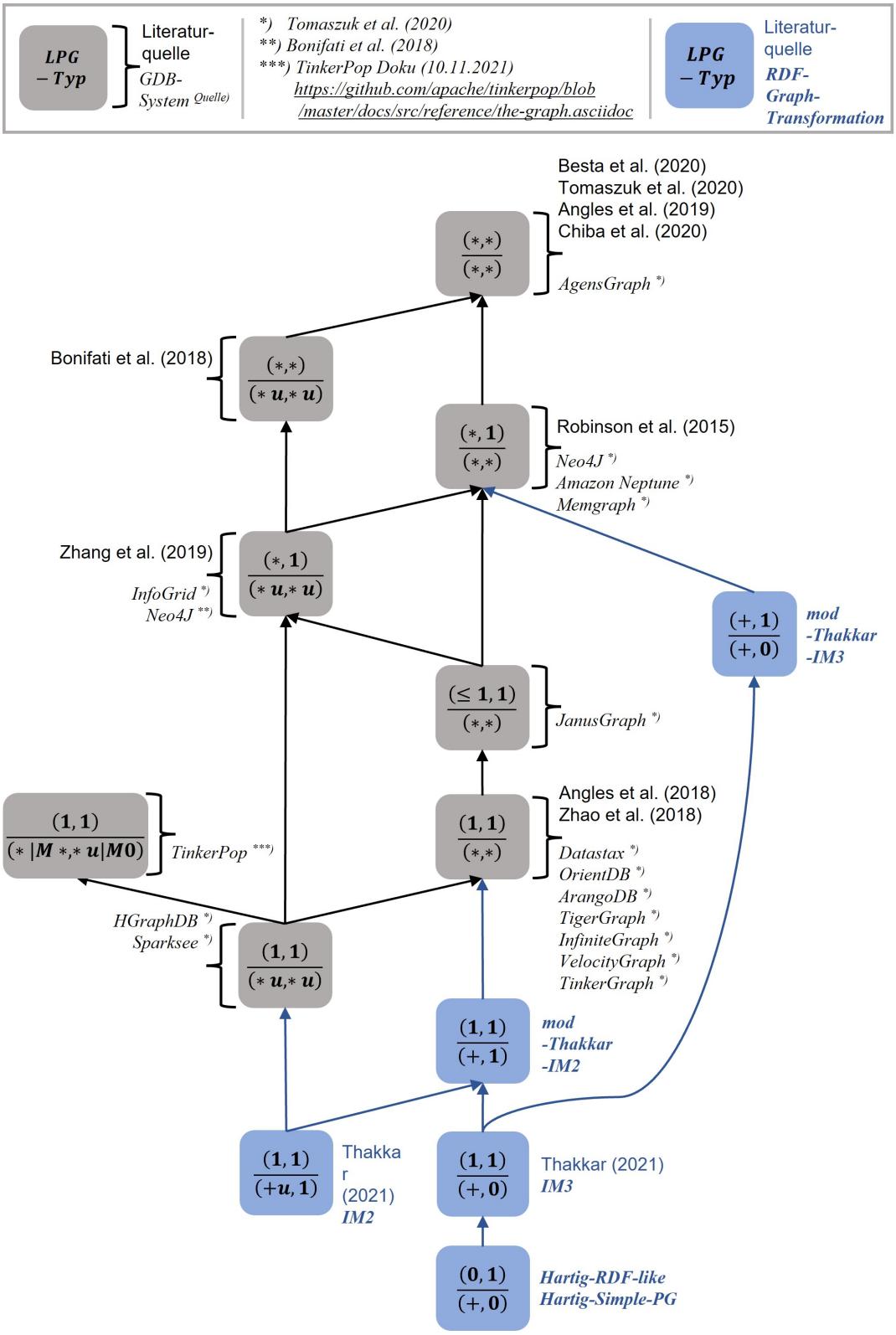

Abb. 8.1: Hierarchie von (M)LP-Graph-Typen gem. Abb. 3.4, S. 26, erweitert um die Typen der Ergebnisgraphen der RDF-Transformationen aus Kap. 7.

eingeordnet. Hieraus lässt sich z.B. entnehmen, welche dieser Ergebnisgraphen in welchem konkreten Graphdatenbanksystem als Datenmodell verwendet werden kann.

Dieser Bericht erfasst einen Zwischenstand von *work in progress*. Neben einer noch zu vertiefenden vergleichenden Betrachtung der vorgestellten Transformationen ist die Ergänzung um weitere relevante, interessante, z.T. bereits wissenschaftlich diskutierte Aspekte und Fragestellungen wünschenswert.

Mit Blick auf RDF-Graphdatenbanken sind dies u.a.:

- Welche Graphstrukturen sind für die Repräsentation der in RDF unterstützten Container-Strukturen (rdf:List, rdf:Bag) notwendig? (siehe Virgilio (2017))
- Wie können benannte RDF Graphen (*named graphs*) bzw. ganze RDF Datasets adäquat formalisiert werden? (siehe Tomaszuk and Szeremeta (2018))
- Wie sollten in RDF Graphen vorhandene reifizierte Statements sinnvoll in Labeled-Property-Graphen abgebildet werden? (siehe Hayes (2004), Zhao et al. (2018))
- Durch welche Art formaler Graphen können RDF*-Graphen geeignet repräsentiert werden? (siehe Hartig (2014))
- Wie können RDF-Schemata bei der Transformation von RDF-Graphen in Labeled-Property-Graphen berücksichtigt werden? (siehe Thakkar (2021),
- Wie können RDF-Schema-Graphen als Schema in Labeled-Property-Graphen dargestellt werden?[2]

Hinsichtlich der Graphmodelle von Labeled-Property-Graphdatenbanken sind bspw. zu nennen:

- Wie lassen sich Labeled-Property-Graphen als RDF-Graphen darstellen?
- Welche Auswirkung auf mögliche Transformationen in RDF-Graphen haben die spezifischen Labeled-Property-Graph-Typen (i.S. unseres Typisierungsschemas)?

Schließlich ist die Frage von Bedeutung, in welcher Weise Transformationen zwischen den Graphmodellen geeignet *konfigurierbar* gestaltet werden können (vgl. z.B. *G2GML* von Chiba et al. 2020a & 2020b) oder *rdf2neo* von Brandizi et al. 2018). Die in diesem Bericht vorgestellten direkten Transformationen haben den Vorteil, dass sie generisch sind und keine besonders einschränkenden Annahmen hinsichtlich der RDF-Graphen treffen. Gleichzeitig sind sie damit unflexibel: Es werden alle Typen zu Labeln oder alle Typen durch spezifische Propertys erfasst; es werden alle Literal-Tripel zu Kanten oder sie werden alle auf Propertys abgebildet. In der Praxis wird es jedoch von den vorkommenden domänenspezifischen Datatype Propertys abhängen, ob sie auf eine Property oder eine Kante eines LP-Graphen abgebildet werden. Letzteres könnte insb. sinnvoll sein, wenn der Wertebereich einer Datatype Property ein Datentyp ist, dessen Werte untereinander als Graph vernetzt werden können[3].

Auf die zukünftige Interoperabilität von Graphdatenbanksystemen hat die beabsichtigte Standardisierung des Labeled-Property-Graph-Modells (sowie ergänzender Schemata) wesentlichen Einfluss. Dies betrifft nicht nur den Austausch von Daten zwischen LP-Graphdatenbanken, sondern in besonderer Weise auch die Verträglichkeit des RDF-Graph-Modells mit diesem standardisierten LP-Graph-Modell. Ein tiefgehendes formales Verständnis der Graphmodelle ist ein wichtige Basis, um Möglichkeiten und Grenzen für die Überbrückung unterschiedlicher Repräsentationen aufzuzeigen.

[2] Hierzu ist die Klärung des Schema-Begriffs für Labeled Property-Graphen relevant. Siehe Hinweise bzgl. aktueller Aktivitäten in Kap.1.

[3] Die ist z.B. gegeben, wenn die Werte Begriffe einer Taxonomie sind.

Literatur

Allemang, D. and J. Hendler (2011). RDFS-plus. In D. Allemang and J. Hendler (Eds.), *Semantic Web for the Working Ontologist (Second Edition)* (Second Edition ed.)., Chapter 8, pp. 153–185. Boston: Morgan Kaufmann.

Angles, R. (2018). The property graph database model. In D. Olteanu and B. Poblete (Eds.), *Proceedings of the 12th Alberto Mendelzon International Workshop on Foundations of Data Management, Cali, Colombia, May 21-25, 2018*, Volume 2100 of *CEUR Workshop Proceedings*. CEUR-WS.org.

Angles, R., H. Thakkar, and D. Tomaszuk (2019). RDF and property graphs interoperability: Status and issues. In A. Hogan and T. Milo (Eds.), *AMW*, Volume 2369 of *CEUR Workshop Proceedings*. CEUR-WS.org.

Angles, R., H. Thakkar, and D. Tomaszuk (2020a). Directly mapping RDF databases to property graph databases. *arXiv*. v2.

Angles, R., H. Thakkar, and D. Tomaszuk (2020b). Mapping RDF databases to property graph databases. *IEEE Access 8*, 86091–86110.

Beckett, D. (2014). RDF 1.1 N-triples - a line-based syntax for RDF graph. http://www.w3.org/TR/2014/REC-n-triples-20140225/ letzter Zugriff 2021-09-25.

Beckett, D., T. Berners-Lee, E. Prud'hommeaux, and G. Carothers (2014). *RDF 1.1 Turtle - Terse RDF Triple Language*. W3C. http://www.w3.org/TR/2014/REC-turtle-20140225/ letzter Zugriff 2021-09-25.

Besta, M., E. Peter, R. Gerstenberger, M. Fischer, M. Podstawski, C. Barthels, G. Alonso, and T. Hoefler (2020). Demystifying graph databases: Analysis and taxonomy of data organization, system designs, and graph queries. *CoRR abs/1910.09017v4*.

Blumauer, Andreas und Nagy, H. (2020). *The Knowledge Graph Cookbook*. monochrom.

Bonifati, A., G. Fletcher, H. Voigt, and N. Yakovets (2018, October). Querying graphs. *Synthesis Lectures on Data Management 10*(3), 1–184.

Brandizi, M., A. Singh, and K. Hassani-Pak (2018). Getting the best of linked data and property graphs: rdf2neo and the knetminer use case. In *SWAT4LS*.

Bray, T., D. Hollander, A. Layman, R. Tobin, and H. S. Thompson (Eds.) (2009). *Namespaces in XML 1.0 (Third Edition)*. W3C. https://www.w3.org/TR/xml-names/ letzter Zugriff 2022-02-06.

Brickley, D. and R. Guha (2014). RDF schema 1.1. http://www.w3.org/TR/2014/REC-rdf-schema-20140225/ letzter Zugriff 2021-09-25.

Cheng, G., Y. Zhang, and Y. Qu (2014). Explass: Exploring associations between entities via top-k ontological patterns and facets. In P. Mika, T. Tudorache, A. Bernstein, C. Welty, C. Knoblock, D. Vrandečić, P. Groth, N. Noy, K. Janowicz, and C. Goble (Eds.), *The Semantic Web – ISWC 2014*, Cham, pp. 422–437. Springer International Publishing.

Chiba, H., R. Yamanaka, and S. Matsumoto (2020a). G2GML: graph to graph mapping language for bridging RDF and property graphs. In J. Z. Pan, V. A. M. Tamma, C. d'Amato, K. Janowicz, B. Fu, A. Polleres, O. Seneviratne, and L. Kagal (Eds.), *The Semantic Web - ISWC 2020 - 19th International Semantic Web Conference, Athens, Greece, November 2-6, 2020, Proceedings, Part II*, Volume 12507 of *Lecture Notes in Computer Science*, pp. 160–175. Springer.

Chiba, H., R. Yamanaka, and S. Matsumoto (2020b). G2GML: graph to graph mapping language for bridging RDF and property graphs. In K. L. Taylor, R. S. Gonçalves, F. Lécué, and J. Yan

(Eds.), *Proceedings of the ISWC 2020 Demos and Industry Tracks: From Novel Ideas to Industrial Practice co-located with 19th International Semantic Web Conference (ISWC 2020), Globally online, November 1-6, 2020 (UTC)*, Volume 2721 of *CEUR Workshop Proceedings*, pp. 363–368. CEUR-WS.org.

Corcho, O., M. Eriksson, K. Kurowski, M. Ojsteršek, C. Choirat, M. van de Sanden, and F. Coppens (2021). *EOSC Interoperability Framework: Report from the EOSC Executive Board Working Groups FAIR and Architecture*. Directorate-General for Research and Innovation, European Commission.

Cyganiak, R., D. Wood, and M. Lanthaler (2014). *RDF 1.1 Concepts and Abstract Syntax*. W3C. http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/ letzter Zugriff 2021-09-03.

Davoudian, A., L. Chen, and M. Liu (2018, April). A survey on nosql stores. *ACM Comput. Surv. 51*(2).

Gandon, F. and G. Schreiber (Eds.) (2014). *RDF 1.1 XML Syntax*. W3C. http://www.w3.org/TR/2014/REC-rdf-syntax-grammar-20140225/ letzter Zugriff 2021-09-25.

Gessert, F., W. Wingerath, S. Friedrich, and N. Ritter (2017, Jul). Nosql database systems: a survey and decision guidance. *Computer Science - Research and Development 32*(3), 353–365.

Group, W. W. (Ed.) (2012). *OWL 2 Web Ontology Language*. W3C. https://www.w3.org/TR/owl2-overview/ letzter Zugriff 2021-09-03.

Harris, S. and A. Seaborne (Eds.) (2013). *SPARQL 1.1 Query Language*. W3C. http://www.w3.org/TR/2013/REC-sparql11-query-20130321/ letzter Zugriff 2021-09-25.

Hartig, O. (2014). Reconciliation of RDF* and property graphs. *CoRR abs/1409.3288*.

Hartig, O., P.-A. Champin, G. Kellogg, and A. Seaborne (Eds.) (2021). *RDF-star and SPARQL-star - Draft Community Draft Report 01 July 2021*. W3C. https://w3c.github.io/rdf-star/cg-spec/2021-07-01.html letzter Zugriff 2021-09-06.

Hartig, O. and B. Thompson (2014). Foundations of an alternative approach to reification in RDF. *CoRR abs/1406.3399*. older version available http://arxiv.org/abs/1406.3399.

Hayes, J. (2004). A graph model for RDF. Technical report, Universität Darmstadt, Universidad de Chile.

Hayes, J. and C. Gutiérrez (2004). Bipartite graphs as intermediate model for RDF. In S. A. McIlraith, D. Plexousakis, and F. van Harmelen (Eds.), *The Semantic Web - ISWC 2004: Third International Semantic Web Conference,Hiroshima, Japan, November 7-11, 2004. Proceedings*, Volume 3298 of *Lecture Notes in Computer Science*, pp. 47–61. Springer.

Hayes, P. and P. F. Patel-Schneider (2014). *RDF 1.1 SEMANTICS*. W3C. https://www.w3.org/TR/rdf11-mt/ letzter Zugriff 2021-09-03.

Heiler, S. (1995, June). Semantic interoperability. *ACM Comput. Surv. 27*(2), 271–273.

Klyne, G. and J. J. Carroll (Eds.) (2004). *Resource Description Framework (RDF): Concepts and Abstract Syntax*. W3C. https://www.w3.org/TR/2004/REC-rdf-concepts-20040210/ letzter Zugriff 2021-09-25.

ORACLE (2021). 17 use cases for graph databases and graph analytics. Technical report, ORACLE.

Peterson, D., S. S. Gao, A. Malhotra, C. M. Sperberg-McQueen, and H. S. Thompson (Eds.) (2012). *XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes*. W3C. https://www.w3.org/TR/2012/REC-xmlschema11-2-20120405/datatypes.html letzter Zugriff 2022-02-06.

Robinson, I., J. Webber, and E. Eifrem (2013). *Graph Databases* (1 ed.). Beijing: O'Reilly.

Robinson, I., J. Webber, and E. Eifrem (2015). *Graph Databases* (2 ed.). Beijing: O'Reilly.

Rodriguez, M. A. (2015). The gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages*, DBPL 2015, New York, NY, USA, pp. 1–10. Association for Computing Machinery.

Schreiber, G. and Y. Raimond (Eds.) (2014). *RDF 1.1 Primer*. W3C. https://www.w3.org/TR/

2014/NOTE-rdf11-primer-20140624/ letzter Zugriff 2021-09-03.

Thakkar, H. (2021). *On Supporting Interoperability between RDF and Property Graph Databases*. dissertation, Universität Bonn.

Tomaszuk, D., R. Angles, and H. Thakkar (2020). PGO: Describing property graphs in RDF. *IEEE Access 8*, 118355–118369.

Tomaszuk, D. and Ł. Szeremeta (2018, September). Named property graphs. In *Proceedings of the 2018 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pp. 173–177. IEEE.

Virgilio, R. D. (2017, May). Smart RDF data storage in graph databases. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pp. 872–881. IEEE.

Zhang, R., P. Liu, X. Guo, S. Li, and X. Wang (2019). A unified relational storage scheme for RDF and property graphs. In *Web Information Systems and Applications*, pp. 418–429. Springer International Publishing.

Zhao, Z., S. K. Han, and J. R. Kim (2018, September). LPG representation of the reification of RDF. *International Journal of Engineering & Technology 7*(3.34), 562.