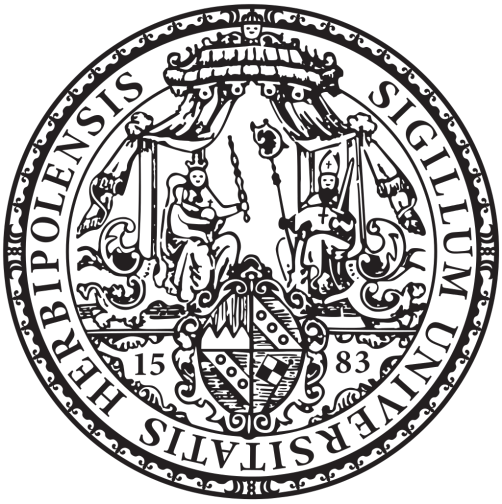
Julius-Maximilians-Universität Würzburg



Integration von Prolog und ClioPatria in Python

– Masterarbeit im Fach Informatik –

Stefan Bodenlos

Eingereicht bei:	Prof. Dr. Dietmar Seipel
Betreuung durch:	Prof. Dr. Dietmar Seipel, M. Sc. Mirco Lukas, Dr. Ludwig Ostermayer
Abgabedatum:	29. September 2017

Inhaltsverzeichnis

Zusammenfassung

Das Ziel dieser Arbeit ist es, einen einheitlichen Ansatz zur intuitiven Integration von Prolog in Python zu präsentieren. In Fallstudien wird festgestellt, dass die bisher existierenden Werkzeuge hierfür nicht ausreichend sind, weil sie einerseits veraltet sind und anderseits eine geringe Benutzerfreundlichkeit bieten – auch weil sie nicht nur unerhebliche Prolog-Kenntnisse voraussetzen. Diese Probleme werden durch das in dieser Arbeit entwickelte Werkzeug CAPPy gelöst, was für Connector Architecture for Prolog and Python steht. Es wurde auf der Vorlage des Tools CAPJa entwickelt, welches eine Integration von Prolog in Java ermöglicht. Hierfür wurden seine Konzepte analysiert und überprüft, inwiefern sie auf eine Integration von Prolog in Python übertragbar sind.

CAPPy ermöglicht es in Python korrespondierender Strukturen zu generieren, die auf dem objektorientiertem Paradigma beruhen und so Prolog-Strukturen leicht verständlich in Python zugänglich machen. Weiterhin wird die Abfragesprache Prolog-Python Query Langauge (Ppql) eingeführt, durch die objektorientierte Abfragen an Prolog in Python realisierbar sind. Sie ist eine interne DSL und verwendet eine Python-ähnliche Semantik. Für die Interaktion mit einem Swi-Prolog-Interpreter wurde ein neues Gateway entwickelt, das mit den neusten Versionen kompatibel ist. Es überführt die Abfrageergebnisse automatisiert in leicht zu verarbeitende Python-Strukturen.

Die Integration von Prolog in Python eröffnet viele neue Entwicklungsmöglichkeiten. Die vorliegende Arbeit widmet sich dabei insbesondere der Verarbeitung von Rdf-Daten. CAPPy ermöglicht dies, ohne dass sein Benutzer eine neue Sprache, wie Sparql, erlernen muss. Hierfür integriert CAPPy die Rdf-Bibliothek ClioPatria. Als Anwendungsbeispiel wird die Yago-Ontologie eingeführt, die eine große Menge an Wissen beinhaltet, welches automatisiert unter anderem aus der Wikipedia extrahiert wurde.

Danksagung

An dieser Stelle möchte ich mich bei allen Personen bedanken, die mich während der Bearbeitung dieser Masterarbeit unterstützt haben.

Dazu möchte ich mich besonders bei meinem Betreuer Prof. Dr. Dietmar Seipel für seine sehr gute Unterstützung bedanken. Seine zahlreichen und immer hilfreichen Anregungen, seine konstruktive Kritik und seine Expertise haben dazu beigetragen, dass diese Thesis in dieser Qualität vollendet werden konnte.

Ein besondere Dank gilt auch M. Sc. Mirco Lukas und Dr. Ludwig Ostermayer, die mich über weite Teile der Arbeit mitbetreut haben. Ihr besonderes Engagement und ihre zahlreichen Ratschläge waren stets eine hervorragende Unterstützung.

Weiterhin möchte ich mich bei meiner Familie und allen Freunden bedanken, die mich während meines Studiums moralisch unterstützt haben und großes Verständnis dafür aufgebracht haben, dass ich aufgrund der Erstellung dieser Abschlussarbeit nur wenig Zeit für sie hatte.

Eigenständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie aus diesen Quellen und Hilfsmitteln wörtlich oder sinngemäß entnommenen Ausführungen als solche kenntlich gemacht habe.

Die Arbeit habe ich bisher oder gleichzeitig keiner anderen Prüfungsbehörde vorgelegt.

Würzburg, 29. September 2017

.....

Stefan Bodenlos

1 Einleitung

In der modernen Softwareentwicklung ist die Kombination verschiedener Technologien ein essentieller Bestandteil. Ein wichtiges Problem hierbei ist, dass sich für die Entwicklung der Verarbeitungs- und der Datenbankkomponente jeweils unterschiedliche Paradigmen entwickelt haben. Die Softwareentwickler müssen deshalb entweder Kenntnisse in beiden Bereichen besitzen oder in getrennten Teams an den jeweiligen Komponenten arbeiten. In beiden Fällen können die unterschiedlichen Paradigmen ein Hindernis in der Entwicklung darstellen. Ziel dieser Arbeit ist es, diese Differenzen zwischen den beiden Ansätzen zu überwinden und einen einheitlichen Ansatz zu präsentieren.

Es existieren bereits viele Ansätze, um populäre Universal-Programmiersprachen, wie Python oder Java, mit populären Datenbanksprachen wie Sql zu verbinden. Allerdings gibt es ein Mangel an benutzerfreundlichen Werkzeugen für die Integration von Prolog in einige wichtige Universal-Programmiersprachen, wie Python. Prolog kann für die Verarbeitung von deduktiven Datenbanken eingesetzt werden. Es ist aber auch eine vollwertige logische Programmiersprache. Viele populäre Universal-Programmiersprachen benutzen imperative und objektorientierte Konzepte. Durch die Integration von Prolog in eine solche Sprache, wird eine neue Möglichkeit eröffnet, solche Technologien zu integrieren, die sich leichter durch logisches Programmieren realisieren lassen. Der Einsatz von Prolog hat sich vorrangig in den Bereichen Intelligente Systeme (insbesondere bei der maschinellen Sprachverarbeitung und dem automatischen Beweisen und Planen), Expertensysteme sowie System- und Netzwerkmanagement bewährt.

Durch das Multipradigmen-Programmieren können Synergieffekte freigesetzt werden. Im Falle der Integration von Prolog in Python bleiben in Python die traditionellen Vorteile erhalten und gleichzeitig ermöglicht dieser neue Ansatz eine einfachere Entwicklung in manchen Bereichen. Eine wichtige Zielsetzung in dieser Arbeit ist, dass die Integration möglichst intuitiv zugänglich sein soll. Dass dies möglich ist, wurde bereits für die Integration von Prolog in Java gezeigt [Ost17]. In dieser Arbeit wird ein vergleichbarer Ansatz für die Integration von Prolog in Python vorgestellt. Dieser Ansatz ist ein Softwareprogramm namens CAPJa, was für Connector Architecture for Prolog and Python steht.

Es ermöglicht eine halb-automatische Integration von PROLOG nach PYTHON. Dazu überführt es PROLOG-Prädikate in eine korrespondierende Struktur in PYTHON, die auf dem objektorientierten Paradigma basiert. Mittels einer neuen Abfragesprache kann ein PYTHON-Programmierer objektorientierte Abfragen an die Datenbank stellen. Dafür übersetzt CAPPY die Abfrage in PROLOG-Strukturen, lässt sie von einem PROLOG-Interpreter auswerten und wandelt das Ergebnis wieder in PYTHON-Strukturen um. Durch die Automatisierung und einem intuitiven Interface soll ein einfacher und produktiver Zugang nach PROLOG ermöglicht werden.

CAPPY eröffnet dabei auch ein Weg, bei dem nicht nur einfache PROLOG-Datenbanken in PYTHON leichter integriert werden können, sondern auch bestehende PROLOG-Prgramme. Diese Arbeit widmet sich dabei im Speziellen der RDF-Bibliothek CLIOPATRIA. Sie stellt viele Funktionalitäten für die Verarbeitung von RDF-Daten bereit. Durch diese transitive Verknüpfung eröffnet CAPPY auch ein einfache Möglichkeit zur Verarbeitung von RDF-Daten.

Die vorliegende Arbeit ist wie folgt strukturiert: In Kapitel 2 werden die grundsätzlichen Begriffe PROLOG, PYTHON, Programmierparadigmen, RDF, YAGO, CLIOPATRIA und domänenspezifische Sprachen eingeführt. Anschließend wird erörtert, welche Vorteile sich aus einer Integration von PROLOG und PYTHON ergeben und welche Problematik dabei grundsätzlich entstehen muss.

In Kapitel 3 werden zwei Fallstudien für die Integration von PROLOG und PYTHON durchgeführt. Sie zeigen auf, welche Vorteile die Integration innehat, warum sie mit derzeitigen Mitteln nicht zufriedenstellend realisiert werden kann und es deshalb den Bedarf für das neue Werkzeug CAPPY gibt.

In Kapitel 4 wird die Machbarkeit dieses neuen Werkzeuges überprüft. Hierfür wird das Tool CAPJA als Prototyp für CAPPY vorgestellt und seine Konzepte herausgearbeitet. Durch eine Analyse der konzeptionellen und technologischen Unterschiede zwischen JAVA und PYTHON wird dargelegt, dass CAPPY in einigen Punkten anders designt werden muss als CAPJA, aber insgesamt als Vorlage genutzt werden kann. Die Funktionsweise von CAPPY wird in einem Entwurf dargelegt. Unter anderem wird auch eine objektorientierte Abfragesprache definiert, welche eine interne domänenspezifische Sprache ist und es einem PYTHON-Benutzer ermöglicht kompakte, intuitive und deklarative Abfragen zu formulieren.

In Kapitel 5 wird auf die Implementierung von CAPPY eingegangen und der strukturelle Aufbau der einzelnen Komponenten erläutert. CAPPY besteht dabei aus drei Kernkomponenten: einer Verbindungskomponente, einer Abbildungskomponente und einer Abfragekomponente. Die Verbindungskomponente heißt PP-

Gateway und stellt eine Schnittstelle für prinzipiell alle Prolog-Interpreter bereit. In dieser Arbeit wird aber nur eine Interaktion mit Swi-Prolog ermöglicht. Die Abfrage-Ergebnisse werden durch einen Zustandsautomaten interpretiert und in einfache Python-Strukturen überführt. Die Abbildungskomponente heißt PPMapping und ermöglicht Prolog-Prädikate und Python-Klassen aufeinander abzubilden. Dazu wird eine bijektive Abbildungsdefinition eingeführt, die beschreibt, wie die einzelnen Strukturen von Prolog nach Python und umgekehrt abgebildet werden. CAPPy unterscheidet zwischen statischer und dynamischer Abbildung. Die erste Form zielt auf eine langfristige und anpassbare Integration ab, während die zweite Form die Strukturen primär temporär und unmittelbar zur Verfügung stellt. Die Abfragekomponente heißt PPQuery und übersetzt Ppql-Abfragen in Prolog-Anfragen. Hierfür wird bestimmt, wie die einzelnen Strukturen der Abfragesprache auf Prolog-Konstrukte übersetzt werden sollen. Weiterhin wird ein Algorithmus für die Übersetzung vorgestellt.

In Kapitel 6 wird die Implementation von CAPPy durch Tests überprüft. Welche weitere Entwicklungsmöglichkeiten sich anbieten, wird ebenso vorgestellt. Bei der Validierung wird gezeigt, dass die Anforderungen an CAPPy erfüllt werden. Als Fazit wird festgestellt, dass mit CAPPy eine einfache Integration von Prolog in Python ermöglicht wird.

2 Einführung in Prolog, Python und Rdf

In diesem Abschnitt werden die zentralen Begrifflichkeiten eingeführt, die das Verständnis dieser Arbeit erleichtern sollen. Das sind zunächst die Programmiersprachen Prolog und Python und die dazugehörigen Programmierparadigmen Logikprogrammierung und Objektorientierung. Weiterhin wird auch der Begriff Rdf erläutert und die Wissensbasis Yago vorgestellt, die auf Rdf-Strukturen basiert. Rdf-Strukturen können durch die Bibliothek ClioPatria in Prolog verarbeitet werden. Als weiterer wichtiger Begriff werden die domänenspezifische Sprachen einführt. Anschließend wird die grundsätzliche Problematik dargelegt, die auftritt, wenn Objekte und Relationen aufeinander abgebildet werden sollen. Zum Schluss werden die Vorteile der Multiparadigmen-Programmierung in Prolog und Python diskutiert.

2.1 Grundbegriffe

In diesem Kapitel werden die Programmiersprachen Prolog und Python eingeführt und aufgezeigt, welche Programmierparadigmen sie jeweils unterstützen. Weiterhin enthält das Kapitel eine kurze Einführung in das Resource Description Frameworks (Rdf), sowie eine Einführung in die Rdf-Bibliothek ClioPatria und der Ontologie Yago. Der letzte Teil des Kapitels widmet sich dem Begriff domänenspezifische Sprachen.

2.1.1 Logikprogrammierung und Prolog

In diesem Abschnitt wird die Logikprogrammierung motiviert und die Programmiersprache Prolog eingeführt und einige seiner Anwendungsgebiete skizziert.

Motivation der Logikprogrammierung. In diesem Abschnitt wird die Logikprogrammierung motiviert. Er stellt eine Zusammenfassung der Arbeit von Pereira und Shieber dar [PS02, S. 1-2].

Es war schon immer eine der Hauptziele in der Entwicklung der symbolischen Logik, den Begriff der logischen Konsequenzen mittels formaler und mechanischer Mitteln zu erfassen. Wenn die Bedingungen von einer bestimmten Problemklasse innerhalb einer geeigneten Logik als Menge von Prämissen formalisiert und ein zu lösendes Problem als ein Satz in der Logik ausgedrückt werden kann, dann kann eine Lösung gegebenenfalls gefunden werden, in dem ein formaler Beweis des Problemsatzes aus den Prämissen konstruiert wird.

Eine Prozedur eines konstruktiven Beweises, die nicht nur die Beweise erzeugt, sondern auch die Werte für die Unbekannten in dem Problemsatz berechnet, kann als ein Rechengerät aufgefasst werden, das die Unbekannten bestimmt. Aus dieser Sichtweise können die Prämissen als ein Programm angesehen werden, der Problemsatz als ein Programmaufruf mit bestimmten Eingabewerten, dessen Ausgabe die gesuchten Belegungen sind, und ein Beweis als eine Berechnung dieses Programms. Dies ist die Grundintuition hinter der logischen Programmierung.

Es reicht jedoch nicht aus eine gültige und vollständige Menge aus Regeln und Schlussfolgerungen sowie eine Prozedur zu haben, die sie systematisch angewendet wird, um ein System der logischen Programmierung zu begründen. Für ein zufriedenstellendes Ergebnis sollte eine Beweisprozedur keine Beweismöglichkeit unüberprüft lassen. Das heißt, die Prozedur terminiert genau dann ohne Beweis, wenn kein Beweis existiert. Des Weiteren hat eine Menge von Prämissen viele Konsequenzen, die für einen Beweis einer gegeben Konsequenz definitiv irrelevant sind. Die Beweisprozedur sollte zielgerichtet sein, sodass irrelevante Konsequenzen vermieden werden. Im Allgemeinen ist eine vollständige Suche bei gleichzeitiger Zielgerichtetheit schwer zu erreichen. In schwächeren logischeren Sprachen ist sie leichter zu realisieren. Die Herausforderung liegt darin, einen guten Kompromiss zwischen der Ausdrucksstärke der logischen Sprache und der Ansprüchen an eine gültige und effiziente Berechnung zu finden.

Ein solcher Kompromiss, von dem auch die Entwicklung der logischen Programmiersprachen abstammt, sind die definiten Klauseln. PROLOG als erste praktische, logische Programmiersprache implementiert sie teilweise.

Prolog. In diesem Abschnitt wird die Programmiersprache PROLOG eingeführt und stellt eine Zusammenfassung der Arbeit von Seipel dar [Sei15].

Die Logikprogrammierung erweitert Datenbanken um komplexere Anfragen, komplexere Datenstrukturen (durch logische Terme), komplexere Regelstrukturen (mit Disjunktion und Constraints) und komplexere Semantikansätze bzw. Kontrollstrukturen (Vgl. [Sei15, S. 9]). Die einfache Handhabung von Termen als Datenstruktur und die mächtige, eingebaute Kontrollstruktur des Backtrackings zeichnet PROLOG im Vergleich zu anderen Programmiersprachen aus. Man kann mit PROLOG auch prozedural, mit Seiteneffekten und mit globalen Variablen programmieren (Vgl. [Sei15, S. 16]). PROLOG ist daher sehr gut zur eingebetteten Datenbankprogrammierung geeignet. Es ist eine deklarative Sprache, mit der Relationen und komplexe Objekte in natürlicher Weise als Fakten oder Terme repräsentiert werden können. Mithilfe deklarativer Regeln können auch Integritätsbedingungen und Inferenzregeln zur Ableitung von Schlussfolgerungen aus der gegebenen Information repräsentiert werden (Vgl. [Sei15, S. 13]). PROLOG ist damit zur Verarbeitung von relationalen Datenbanken und Semantic-Web-Anwendungen ideal.

PROLOG benutzt eine pradikatenlogische Syntax mit einer auf der *SLDNF*-Resolution basierenden Abfrageauswertung. Die *SLDNF*-Resolution ist eine effiziente Methode des Theorembeweisens. Sie ist eine Erweiterung der *SLD*-Resolution, einer vollständigen und korrekten Art des Theorembeweisen für ursprünglich allgemeine Klauseln. Neben den rein logischen Prädikaten besitzt PROLOG auch solche mit Seiteneffekten bzw. solche, die zur Steuerung der Abfrageauswertung dienen. Deswegen handelt es sich nicht um eine reine *SLDNF*-Resolution, sondern um eine Erweiterung dieser (Vgl. [Sei15, S. 28-29]).

Strukturen in Prolog. In PROLOG werden die aus der Prädikatenlogik bekannten Terme und Atome zu dem Konzept der Strukturen vereinheitlicht. Selbst Regeln sind Strukturen. Nach Seipel können die Strukturen in PROLOG wie folgt beschrieben werden (Vgl. [Sei15, S. 106-108]):

Ein PROLOG-Atom ist eine Folge von Buchstaben, Ziffern und dem Unterstrich "_" (beginnend mit einem Kleinbuchstaben), eine Folge von Sonderzeichen oder eine Folge von Zeichen, die in Hochkommata eingebettet sind. Eine Zahl ist eine ganze Zahl oder eine reelle Zahl. Eine Konstante kann ein PROLOG-Atom oder eine Zahl sein. Variablen müssen von Konstanten unterscheidbar sein. Eine Variable ist daher eine Folge von Buchstaben, Ziffern und dem Unterstrich "_" (beginnend mit einem Großbuchstaben) oder lediglich ein Unterstrich "_". Man nennt "_" die anonyme Variable. Man benutzt sie, wenn man sich für eine Argumentposition nicht interessiert, denn jedes Vorkommen von "_" in einem Programm steht für eine andere Variable.

Ein Funktor ist ein PROLOG-Atom. Manche Folgen von Sonderzeichen haben in PROLOG eine vordefinierte Bedeutung als Funktoren. Eine einfache Struktur ist eine Konstante oder eine Variable. Eine komplexe Struktur ist von der Form $\varphi(s_1, ..., s_n)$, mit einem Funktor φ und n Strukturen s_i. Komplexe Strukturen werden also induktiv aus Funktoren und anderen komplexen Strukturen (beginnend mit Variablen und Konstanten) gebildet.

Eine PROLOG-Regel ist eine Struktur der Form `Kopf :- Rumpf` mit dem binären Infix-Funktor `":-"` und den zwei Strukturen `Kopf` und `Rumpf`. Der Funktor `":-"` ist der Regeloperator, der für einen Folgepfeil "←"nach links steht.

Declare Developers' Kit (Declare). DECLARE "ist ein Mehrzweck-Prolog-Softwarepaket; Seine Funktionalität reicht vom (nicht-monotonem) Schließen in disjunktiven deduktiven Datenbanken (...) bis zu Abfragen von XML-Datenbanken, Unterstützung für Softwareentwicklung in PROLOG und JAVA (...)"[Sei17]. Das DECLARE stellt somit eine Reihe von Funktionalitäten bereit, die die Entwicklung von PROLOG-Anwendungen erleichtern.

Verarbeitung von natürlicher Sprache in Prolog. In diesem Teilabschnitt wird skizziert, warum die Programmiersprache PROLOG für die Verarbeitung natürlicher Sprache besonders prädestiniert ist und stellt im Wesentlichen eine Zusammenfassung der Arbeit von Pereira und Shieber (vgl. [PS02, S. 2]).

Fast seit Anbeginn der Entwicklung der logischen Programmiersprachen sind sie eng an das Ziel gebunden, berechenbaren Formalismen zu schaffen, mit denen die syntaktische und semantische Analyse von Sätzen in natürlicher Sprache durchgeführt werden können. Eine der Hauptziele bei der Entwicklung von PROLOG war eine Programmiersprache zu erschaffen, in der die Regeln für die Phasenstruktur und der semantischen Interpretation natürlichsprachlicher Fragen-Antwort-Systeme einfach ausgedrückt werden können.

Phrasenstruktur-Regeln für eine Sprache drücken aus, wie Phrasen eines gegebenen Typs kombiniert werden können, um größere Phrasen in der selben Sprache formen zu können. Zum Beispiel kann eine vereinfachte Phrasenstruktur-Regel für deklarative Sätze beschrieben werden, als ein Satz, der aus einer Nominalphrase (dem Subjekt des Satzes) gefolgt von der Verbalphrase (dem Prädikat des Satzes) besteht.

Diese Regel kann leicht in der Prädikatenlogik ausgedrückt werden:

$$(\forall u, v, w) NP(u) \wedge VP(v) \wedge conc(u, v, w) \implies S(w)$$

Hierbei repräsentiert NP die Klasse der Nominalphrasen, VP die Klasse der Verbalphrasen und S die Klasse der Sätze. Die Konkatenation $conc(u, v, w)$ sagt aus, dass wenn die Zeichenkette v der Zeichenkette u folgt, dass dann daraus ihre Konkatenation w folgt. Der komplette Ausdruck beschreibt also, dass eine beliebige Nominalphrase u und eine beliebige Verbalphrase v zu dem Aussagesatz $w = uv$ konkateniert werden können. Der Ausdruck *logische Grammatik* wurde gebildet, um einen solchen Gebrauch der Logik zu bezeichnen, die grammatikalische Regeln formalisiert.

Viele wichtige Klassen von linguistischen Regeln und Bedingungen kann in eine solche, allgemeine Form gebracht werden, weshalb definite Klauseln in der Sprachanalyse nützlich sind. Da definite Klauseln direkt in PROLOG verarbeitet werden können, bietet PROLOG eine effiziente und direkt berechenbare Realisierung von Grammatiken und Interpretationsregeln.

2.1.2 Die Prolog-Bibliothek ClioPatria für Rdf

In diesem Abschnitt wird das RESOURCE DESCRIPTION FRAMEWORK (RDF) und die darauf basierende Ontologie YAGO eingeführt. Im Anschluss daran wird auf die RDF-Bibliothek CLIOPATRIA eingegangen.

Das Resource Description Framework

Das RESOURCE DESCRIPTION FRAMEWORK (RDF) ist ein Datenmodell für das Web of Data und dem Semantic Web, das Informationen über die Ressourcen im World Wide Web bereit stellt. Es stellt logische Organisationen in Form von Datenstrukturen bereit, die die Repräsentation, den Zugang, die Constraints und die Beziehungen unterstützen. Die Grundeinheit ist der RDF-Tripel, der die Form (s, p, o) hat. Dabei steht s für das Subjekt, o für das Objekt und p für das Prädikat. Er sagt damit aus, dass eine bestimmte Sache (das Subjekt) durch eine bestimmte Eigenschaft (dem Prädikat) mit einem bestimmten Wert (dem Objekt) beschrieben wird. Vgl. [CB14, Kap. 3.2]

Ein Ressource ist ein Gegenstand in einem Diskursuniversum. Ein Problem hierbei ist, wie eine gemeinsame Ressource mehrerer Quellen identifiziert werden kann. Hierfür wird der UNIFORM RESOURCE IDENTIFIER (URI) genutzt. Eine URI ist ein globaler Bezeichner für die Ressoursen im Netz. Die Syntax der URIs ist denen der URLs sehr ähnlich. Zwei Quellen im Netz können auf die selbe Ressource mit der gleichen URI referenzieren. Der Vorteil hiervon ist, dass in den URIs be-

Listing 2.1: Beispiel-Relation in Yago

```
1 AlbertEinstein hasWonPrize Nobelpreis
```

reits nützliche Zusatzinformationen, wie den Servernamen oder der Dateiname, entnommen werden können. Vgl. [CB14, Kap. 3.2]

Die Spezifikation von Rdf ist Graphen-basiert. Es kann als Grundlage dienen, um Ontologien zu erstellen (Vgl. Web Ontology Language). Nach Paulheim [Pau11, S. 28] existieren verschiedene Definitionen für den Begriff Ontologie in der Informatik. Häufig versteht man unter diesem Begriff eine explizite Spezifikation einer Konzeptualisierung. Eine genauere Definition besagt, dass eine Ontologie eine logische Theorie ist, die eine Konzeptualisierung explizit und partiell beschriebt. Nach einer noch präziseren Definition ist eine Ontologie eine Menge logischer Axiome, die entwickelt wurden, um die beabsichtigte Bedeutung eines Wortschatzes zu beschreiben. Ontologien könne Definitionen über Kategorien und Instanzen samt ihre Beziehung untereinander enthalten.

Die Ontologie Yago

Yago ist eine Ontologie mit hoher Abdeckung und Qualität. Sie baut auf dem Konzept von Entitäten und Relationen auf und bestand im Juni 2017 aus mehr als zehn Millionen Entitäten und 120 Millionen Fakten [Inf]. Im Folgenden wird die Arbeit von Suchanek et al. zusammengefasst (vgl. [SKW07]).

Die Struktur des Yago-Modells. Das Yago-Modell ist eine Erweiterung des Rdf-Modells. Es bleibt aber trotzdem mit Rdf kompatibel. Es kann Relationen zwischen Fakten und Relationen ausdrücken. Alle Objekte werden als Entitäten repräsentiert, wobei zwei Entitäten miteinander in Beziehung stehen können. Steht z. B. die Entität *Albert Einstein* mit der Entität *Nobelpreis* in der *hasWonPrize*-Relation, so notiert man dies im Yago-Modell wie in Listing 2.1.

Im Yago-Modell sind auch Wörter (erkennbar durch die Anführungszeichen) Entitäten. Dadurch können auch Wörter auf eine bestimmte Entität verweisen, wie im Beispiel-Listing 2.2. Ähnliche Entitäten werden in Klassen gruppiert, wobei jede Entität zumindest die Instanz einer Klasse sein muss. Dies wird durch die *type*-Relation ausgedrückt, wie in Listing 2.3.

Listing 2.2: Wort als Yago-Entität

```
"Einstein" means AlbertEinstein
```

Listing 2.3: *type*-Relation in Yago

```
AlbertEinstein type physicist
```

Auch Klassen sind Entitäten. Daher ist jede Klasse eine Instanz von der Klasse *class*. Durch die *subClassOf*-Relation kann eine taxonomische Hierarchie von Klassen gebildet werden, wie im Beispiel-Listing 2.4.

Selbst Relationen werden als Entitäten aufgefasst. Dadurch können Eigenschaften der Relationen ausgedrückt werden, wie zum Beispiel-Listing 2.5

Ein Tripel mit zwei Entitäten und einer Relation hierüber nennt man einen Fakt, wobei die Entitäten die Argumente des Faktes genannt werden. Jedem Fakt wird ein Bezeichner gegeben, der wiederum eine Entität ist. Wenn der Fakt *(AlbertEinstein, hasWonPrize, Nobelpreis)* den Bezeichner #1 hat, so kann man in Yago wie im Listing 2.6 ausdrücken, wo der Fakt gefunden wurde.

Entitäten, die weder Fakten noch Relationen sind, heißen gewöhnliche Entitäten. Gewöhnliche Entitäten, die keine Klassen sind, heißen Individuen. Manche Fakten benötigen mehr als zwei Argumente. Um solche n-stelligen Relationen auszudrücken, muss ein Hauptpaar zweier Argumente bestimmt werden Der Fakt, dass Einstein den Nobelpreis im Jahre 1921 gewonnen hat, kann durch ein Hauptpaar dargestellt werden, wie in Listing 2.7 Die anderen Argumente können dann als Relationen über dem Hauptpaar und dem Argument aufgefasst werden, siehe Listing 2.8

Herkunft der Daten. Die Fakten in Yago wurden automatisiert aus der Wikipedia extrahiert und mittels WordNet vereinheitlicht.

Wikipedia Die Wikipedia ist eine mehrsprachige, freie und Web-basierende Online-Enzyklopädie. Jeder Artikel in der Wikipedia ist eine einzelne Webseite und

Listing 2.4: *subClassOf*-Relation in Yago

```
physicist subClassOf scientist
```

Listing 2.5: Relation als Entität in Yago

```
subclassOf type transitiveRelation
```

Listing 2.6: *foundIn*-Relation in Yago

```
#1 foundIn http://www.wikipedia.org/Einstein
```

beschreibt in der Regel ein einziges Thema. Sie enthält im Mai 2017 mehr als 5,4 Millionen englischsprachige Artikel [Wik]. Den meisten Artikel wurde eine oder mehrere Kategorien von Hand zugewiesen. Die Kategorisierung der Wikipedia-Seiten und ihre Link-Struktur sind als Sql-Tabellen erhältlich.

WordNet WordNet ist ein semantisches Lexikon für die englische Sprache. Es unterscheidet zwischen Wörtern, wie sie wörtlich in einem Text auftauchen, und der eigentlichen Bedeutung der Wörter. Ein Menge von Wörtern, die sich eine Bedeutung teilen, heißt *Synset*. Wörter mit mehreren Bedeutungen gehören zu mehreren *Synsets*. Zwischen den *Synsets* stellt WordNet die Beziehungen Ist-Unterkonzept, Ist-Oberkonzept, Ist-Teil-Von und Besteht-Aus bereit.

Für die Extraktion wurden regelbasierte und heuristische Methoden kombiniert. Die einzelnen Relationen wurden wie folgt extrahiert:

Type Die Individuen wurden aus der Wikipedia extrahiert, wobei jeder Titel eines Artikel ein Kandidat für ein Individuum ist. Um die Klassen der Individuen zu bestimmen, wurde das Kategorie-System der Wikipedia ausgenutzt. Dabei beschreiben nur die *konzeptuellen Kategorien* wirklich die Klasse einer Entitäten einer Seite. Die anderen sind irrelevant. Kategorien für administrative Zwecke oder für relationale Informationen wurden aufgrund der geringen Zahl von Hand ausgeschlossen. Kategorien, die lediglich die Thematik eines Artikels beschreiben, wurden durch eine Computer-lingustische Methode ermittelt und ausgeschlossen. Da Artikel, die keine Individuen beschreiben, keine konzeptuellen Klassen besitzen, wird hierdurch auch zugleich die Menge der Individuen in Yago definiert.

Listing 2.7: Hauptpaar einer mehrstelligen Relation in Yago

```
#1 foundIn http://www.wikipedia.org/Einstein
```

Listing 2.8: Ergänzung einer mehrstelligen Relation in Yago

```
1 #1 time 1921
```

SubClassOf Die Kategorien in der Wikipedia werden durch gerichtete, azyklische Graphen organisiert, wobei sie bloß die thematische Struktur der Seiten reflektieren. Daher werden nur die untersten Kategorien genommen und mithilfe von WordNet die Klassenhierarchie bestimmt. Dabei wird jedes *Synset* (ausgenommen Eigennamen) in WordNet eine Klasse in Yago. In manchen Fällen ist die Entität sowohl im WordNet als auch in der Wikipedia enthalten. Dann wird immer der WordNet-Fall übernommen, da es in der Wikipedia auch Kategorien für Eigennamen gibt. Die *SubClassOf*-Relation wurde entsprechend der Ist-Unterbegriff-Relation im WordNet etabliert. Einige Ausnahmen wurden von Hand geändert.

Means Die *means*-Relation wurde aus dem WordNet anhand der *Synsets* entnommen. Dazu wurde für jedes *Synset* eine Klasse eingeführt. Die *means*-Relation wird für jedes Wort im *Synset* und der entsprechenden Klasse etabliert. Aus der Wikipedia wurde die *means*-Relation anhand der Umleitungen extrahiert. Diese Umleitungen werden in der Wikipedia eingesetzt, um von Artikeln mit alternativen Namen zu den eigentlichen umzuleiten.

Weitere Relationen Die Relationen *bornInYear*, *diedInYear*, *establishedIn*, *locatedIn*, *writtenInYear*, *politicianOf* und *has- WonPrizewurden* anhand der entsprechenden Wikipedia-Kategorien extrahiert.

Die Wissensbasis Yago besitzt daher eine höhere Qualität als WordNet allein, da sie einerseits auch Wissen über Personen, Produkte, Organisationen etc. mitsamt ihren semantischen Beziehungen enthält und andererseits um Größenordnungen mehr Fakten beinhaltet. In einer empirischen Evaluation ergab sich eine Korrektheit der Fakten von ungefähr 95 %. Yago kann durch weitere Techniken der Informationsextraktion erweitert werden.

Erweiterungen von Yago. Yago wurde seit seiner ersten Veröffentlichung im Jahre 2007 immer wieder in neuen Versionen veröffentlicht und dabei um mehrerer Funktionalitäten erweitert, die in diesem Abschnitt kurz vorgestellt werden.

In der Version von 2008 wurden auch die Infoboxen in der Wikipedia zur Wissensextraktion verwendet, Techniken zu Qualitätskontrolle eingefügt und eine neue Abfragesprache definiert. Infoboxen sind deswegen so interessant, weil sie viele

Informationen über das Thema des Artikels liefern. Manuell wurden einige häufig eingesetzte Attribute identifiziert, für die jeweils eine eigene Relation eingeführt wurde. Grundsätzlich wird aus jeder Zeile der Infobox ein Fakt generiert. Dabei muss beachtet werden, dass manche Bedeutungen der Attribute von dem Typ der Infobox abhängt. So ist die Länge bei einem Auto eine räumliche Größe, während sie in einem Musikstück eine zeitliche Größe ist (Vgl. [SKW08]).

In Yago2 wurde das Framework für die Faktenextraktion erweitert, sodass Infoboxen, Listen, Tabellen, Kategorien und reguläre Muster im Freitext unterstützt werden. Auch eine schnelle und einfache Spezifikation neuer Extraktionsregeln wurde eingeführt. Eine erweiterte Wissensrepräsentation beachtet auch Zeiten und Örtlichkeiten. Dazu wurde auch Wissen aus der Geographie-Datenbank GeoNames extrahiert. Durch eine neu Form der Repräsentation können raum-zeitliche Informationen leicht abgefragt werden (Vgl. [HSBW13]).

In Yago3 wurde nicht mehr ausschließlich die englischsprachige Wikipedia genutzt, sondern auch anderssprachige Ausgaben der Wikipedia. Dabei wurde aber nicht für jede Sprache eine einzelne Wissensabis erstellt, sondern eine kohärente Faktensammlung aus den unterschiedlichen Quellen erschaffen. Dazu mussten unter anderem viele Attribute in den Infoboxen in den verschieden Sprachen auf eine kanonische Darstellung abgebildet werden. Es musste auch die Taxonomie über die verschieden Sprachen vereinheitlicht werden. Dadurch ist eine Wissensbasis entstandenen, die viel weniger Fehler enthält, einheitlich ist und viele Millionen zusätzlicher Fakten, im Vergleich zur Vorgängerversion, enthält.

Die Rdf-Bibliothek ClioPatria

ClioPatria ist eine Rdf-Bibliothek für Swi-Prolog. Dieser Abschnitt führt es ein und ist, soweit nicht anders gekennzeichnet, eine Zusammenfassung der Arbeit von Wielemaker et al. (Vgl. [WBHv16]).

Das Semantic Web bietet ein gemeinsames Framework an, mit dem Daten über Anwendungen, Unternehmen und Gemeinschaften hinweg geteilt und wiederverwendet werden können [Con]. Dabei wird im Semantic Web häufig die dreigliedrige Architektur mit der Aufteilung Speicher, Anwendungslogik und Präsentation eingesetzt. Jedoch kann bei einer solchen Architektur nicht das vollständige Potential des Semantic Webs ausgeschöpft werden. Zum Einen liegt das an der Trennung zwischen Speicher und Anwendungslogik, denn dadurch müssen zwei verschiedene Programmiersprachen miteinander verknüpft werden, da es für die beiden Bereiche jeweils unterschiedliche Spezial-Sprachen gibt. Hierdurch wird die Lesbarkeit des Programmcodes beeinträchtigt, da der Programmierer beide Programmierspra-

chen beherrschen muss, um ihn vollständig verstehen zu können. Weiterhin wird durch die Verwendung zweier Programmiersprachen auch die Berechnung verlangsamt, weil es zusätzliche Berechnungsschritte bedarf, um den Speicher anzusprechen. Zum Anderen unterstützen auch viele Sprachen, die in der Anwendungslogik benutzt werden, keine Logikprogrammierung, also keine elegante Möglichkeit komplexe, regelbasierte Auswertungen einzusetzen. Diese Probleme können durch ClioPatria gelöst werden.

Eigenschaften von ClioPatria. ClioPatria ist ein Prolog-basierendes Framework, dessen Augenmerk hauptsächlich auf dem Prototypenbau liegt, einer Methode aus der Softwareentwicklung, bei der schnell ein lauffähiges Programm erstellt werden soll, damit der Kunde leichter den Vorschlag des Entwicklers bewerten kann. Hierzu enthält es folgende Features:

- Bibliotheken, die alle standardmäßigen Rdf-Austauschformate unterstützen
- Einen stabilen Regelformalismus und erleichterter Prototypenbau durch die Programmiersprache Prolog
- Ein Webentwicklungs-Framework für Rdf-basierende Anwendungen
- Einer Komponente, dass die Rdf-Tripel effizient und stabil speichert

Vergleich von Abfragen in Prolog und Sparql. Standardisierte Abfragesprachen haben eine beschränke Ausdrucksstärke für Softwaredaten. Deshalb ist immer eine Kombination aus serverseitiger Abfrageauswertung und clientseitiger Nachbearbeitung nötig. Während die Suche nach Literalen in Sparql auf reguläre Ausdrücke und numerische Bedingungen beschränkt ist, stellt ClioPatria eine ausdrucksstärkere Programmierumgebung bereit, denn Prolog bietet eine gute Unterstützung für die Verarbeitung natürlicher Sprachen. ClioPatria kann, sofern die Option aktiviert ist, für alle Literale einen Volltext-Index bereitstellen, ggf. mit Reduzierung auf einen Wortstamm (*Stemming*) und ähnlich klingenden Wörtern (Ergebnisse des *Metaphone*-Algorithmus), und so eine schnelle Suche nach bestimmten Tokens ermöglichen. Dadurch können auch komplexe Operationen, die eine Verarbeitung von natürlicher Sprache benötigen, ausgewertet werden.

Die einzelnen Prolog-Regeln können als Abstraktionen von mehreren Rdf-Tripeln aufgefasst werden. Prolog-Regeln anstatt Sparql-Abfragen zu nutzen führt zu einigen Vorteilen. Zunächst haben Prolog-Regeln immer einen Namen und können so einfach bei weiteren Abfragen wiederverwendet werden. Dies ist ein effektiver Weg, um lange und komplexe Abfragen, wie in Sparql, zu umge-

Listing 2.9: Rdf-Tripel in ClioPatria

```
1 rdf(Subject, Predicate, Object)
```

hen. Für einen Entwickler erleichtert dies das Erstellen von komplexen Funktionseinheiten deutlich, denn kleinere Blöcke lassen sich viel einfacher testen und verifizieren als größere. Des Weiteren ist bereits die Wahrscheinlichkeit in kleinen Blöcken geringer, dass der Entwickler überhaupt Fehler in sie einbaut. Auch kann es umständlich sein, einen bestimmten Teil einer komplexen, bereits existierenden Sparql-Abfrage zu extrahieren. Denn bauen die Blöcke für eine komplexe Abfrage stets inkrementell aufeinander auf, dann kann der Teil, der wiederverwendet werden soll, einfacher lokalisiert und wiederverwendetet werden. Zusätzlich haben Prolog-Regeln den Vorteil, dass sie nicht nur für reine Abfragen genutzt werden können, sondern auch zum Filtern, indem Variablen in einer Abfrage durch Atome ersetzt werden. Es können auch externe Quellen durch relationale Datenbanken oder andere Prolog-Relationen einfach in eine Rdf-Abfrage miteinbezogen werden. Prolog besitzt darüber hinaus auch mächtigere Kontrollstrukturen, die mit Rekursion und Constraints umgehen können.

Es existieren einige realistische Abfragen, die in Sparql nicht ausgedrückt werden können. Zum Beispiel können Rdf-Tripel nur bis zu einer, in der Abfrage fest vorgegebenen Länge, verkettet werden. Möchte man beispielsweise wissen, über welchen Pfad eine gesuchte Zeichenkette von einer Ressource eines gegebenen Typs erreicht werden kann, so muss bei der Formulierung der Abfrage von einer festen Pfadlänge ausgegangen werden, über der die einzelnen Literale verknüpft werden. In Prolog existieren solche Einschränkungen nicht. Es ist problemlos möglich, nach einer beliebigen Pfadlänge zu suchen.

Technische Realisierung von ClioPatria. ClioPatria wandelt jede Sparql-Abfrage in Prolog-Konstrukte um, in dem Kontrollprimitiven oder (Meta-) Prädikate eingesetzt werden. So kann beispielsweise ein *Union* in Sparql als Disjunktion in Prolog interpretiert werden. Auch viele weitere Sparql-Konzepte lassen sich so in Prolog-Konstrukte übersetzen.

Ein Rdf-Tripel wird in ClioPatria durch eine Relation repräsentiert, siehe Listing 2.9. Der Uniform Resource Identifier wird durch ein einfaches Prolog-Atom repräsentiert. Literale werden durch einen Prolog-Term ausgedrückt, der die lexikalische Form, den Datentyp oder den Sprach-Tag repräsentiert. Präfixe können wie in Listing 2.10 deklariert werden. Nachdem eine solche Deklaration

Listing 2.10: Präfix-Definition in ClioPatria

```
1 rdf_register_prefix(Prefix, URI)
```

durchgeführt wurde, können die Uris direkt mit `prefix:localname` abgekürzt werden.

Obwohl auch die Rdf-Tripel direkt durch dynamische Prolog-Prädikate ausgedrückt werden könnten, wurde dies durch ein externes Tool durchgeführt, das Triple Store genannt wird. Denn Rdf-Tripel haben nur eine sehr eingeschränkte Form, da sie keinen Regelrumpf haben, die Subjektive und Prädikate stets Uris sind und es nur eine beschränkte Anzahl an Grundtermen für die Objekte gibt. Der Triple Store wurde in der Programmiersprache C geschrieben und besitzt eine auf Rdfs optimierte Speicherstruktur. Im Gegensatz zu einer allgemeinen Text-Indexierung hat der Triple Store daher die Vorteile, dass er weniger Speicher verbraucht, die Zugriffszeiten schneller sind und es weniger Synchronisationsprobleme gibt.

2.1.3 Python und Objektorientiertes Programmieren

Python ist eine universelle, interpretierbare, interaktive und objektorientierte Höhere Programmiersprache. Es besitzt eine hohe Mächtigkeit und eine klare Syntax. Seine große Standard-Bibliothek stellt unter Anderem eine Unterstützung für Internetprotokolle, Verarbeitung von Zeichenketten, Softwaretechnik und Betriebssysteminterfaces bereit. Python ist auch in den meisten gängigen Betriebssystemen enthalten (Vgl. [vPa, S. 1-2]). Python unterstützt viele Programmierparadigmen und deswegen kann man Python bereits als eine Mulitparadigmen-Programmiersprache betrachten. Konzepte der Logikprogrammierung unterstützt es jedoch nicht.

Objektorientiertes Programmieren. Gabbrielli und Martini [GM10, S. 282-297] betrachten folgende Konzepte als wesentlich für das objektorientierte Programmieren:

Objekte Die Grundstruktur von objektorientierten Sprachen ist das Objekt. Es enthält sowohl die Daten als auch die Operationen, die sie manipulieren und eine Schnittstelle nach außen darstellen, durch die auf das Objekt zugegriffen werden kann.

Klasse Eine Klasse ist ein Modell für eine Menge von Objekten. Sie definiert, welche Daten sie haben und bestimmt die Namen, Signaturen, Sichtbarkeiten und die Implementierung für jede Methode. Jedes Objekt gehört zu mindestens zu einer Klasse.

Datenkapselung In jeder Klasse gibt es mindestens zwei Sichten: `private` und `public`. Die `private`-Sicht gilt innerhalb eine Klasse und in ihr ist alles sichtbar und alles zugreifbar. In der `public`-Sicht sind jedoch nur die explizit freigegeben Informationen sichtbar.

Subtypen Eine Klasse kann auf natürliche Art und Weise mit einer Menge von Objekten korrespondieren, welche Instanzen der selben Klasse sind. In ungetypten Sprachen, wie PYTHON, basiert diese Korrespondenz nur auf Konventionen und ist implizit.

Vererbung Die Vererbung ist ein Mechanismus, der die Definition von neuen Objekten ermöglicht, in dem bereits existierende Objekte wiederverwendet werden. Dadurch kann ein bereits geschriebener Code wiederverwendet und auch erweitert werden. Die Klasse, von der geerbt wird, heißt *Oberklasse* und die Klasse, die erbt, heißt *Unterklasse*. Wird die Implementierung einer Klasse modifiziert, so wird die Änderung automatisch an alle *Unterklassen* übertragen.

Python als objektorientierte Programmiersprache. PYTHON besitzt Elemente der Objektorientierung. In manchen Punkten verletzt es aber die zuvor genannte Definition:

Datenkapselung In PYTHON kann prinzipiell immer direkt auf ein Attribut eines Objektes zugegriffen werden. Es existieren keine Sichtbarkeiten, wie `public` oder `private`. Mittels einer Konvention werden die Attribute, die von außen nicht manipuliert werden sollen, gekennzeichnet.

Typ-Hinweise Attribute in PYTHON können keinen festen Typen zugeordnet werden. Als Alternative bietet es Typ-Hinweise an. Dies ist ein natives Konstrukt in PYTHON, bei dem der Benutzer einer Funktion oder Methode darauf hingewiesen wird, welchen Typ ein Parameter bzw. der Rückgabewert entsprechen soll. Die Einhaltung der Typ-Hinweise obliegt dem Benutzer selbst. Indem im Konstruktor Typ-Hinweise benutzt werden, können die Typen der Attribute indirekt gekennzeichnet werden.

Attributmenge Ein Objekt kann sich zur Laufzeit selbst dynamisch neue Attribute geben. Darüber hinaus kann sogar von beliebiger externer Stelle ein Objekt

um beliebig viele neue Attribute erweitert werden. In PYTHON haben damit nicht alle Objekte einer Klasse notwendigerweise die selbe Attributmenge.

2.1.4 Domänenspezifische Sprachen

Nach Fowler [Fow10, Chap. 2.1] ist eine domänenspezifische Sprache (DSL) eine Programmiersprache, die eine beschränkte Ausdrucksstärke besitzt und auf eine bestimmte Domäne ausgerichtet ist. In diesem Abschnitt wird auf diesen Begriff genauer eingegangen. Es ist im Wesentlichen eine Zusammenfassung von der Arbeit von Fowler [Fow10].

Da DSLs Programmiersprachen sind, werden sie von Menschen eingesetzt, um einen Rechner Instruktionen zu erteilen. Sie sind dabei so designt, dass sie leicht von Menschen verstanden werden können, aber dennoch auf einem Rechner ausführbar sind. Weiterhin sollen sie für Menschen beherrschbar sein. Die Ausdrucksstärke rührt dabei nicht nur von einzelnen Ausdrücken her, sondern vielmehr von der Art und Weise, wie sie zusammengesetzt werden. Im Gegensatz zu den universellen Programmiersprachen, besitzen DSLs nicht alle möglichen, im Allgemeinen nützliche, Funktionalitäten. Sie schränken sich auf die nötigsten ein, die für die gegebene Domäne wirklich benötigt werden. Hierdurch sollen sie leichter erlernbar sein. Diese Beschränkungen macht nur auf einer kleinen Domäne Sinn. Man kann mit einer DSL kein komplettes Softwaresystem bauen, sondern lediglich ganz bestimmte Aspekte davon. Dieser Fokus auf die Domäne wiegt die Beschränkung der Sprache wieder auf (Vgl. [Fow10, Chap. 2.1]).

Fowler unterteilt die DSL in drei Hauptkategorien ein:

Externe DSLs Das sind Sprachen, die von der Hauptsprache einer gegebenen Anwendung unabhängig ist. Sie besitzen in der Regel eine eigene Syntax, können aber auch die Syntax einer anderen Sprache benutzen (wie XML). Die Anweisungen in einer externen DSL werden normalerweise von der Anwendung geparst.

Interne DSLs Eine interne DSL ist ein bestimmter Weg, wie eine allgemeine Sprache genutzt wird. Der Code in einer internen DSL ist damit auch ein gültiger Code in der allgemeinen Sprache. Die Sprachdefinition erstreckt sich aber nur auf eine Teilmenge von den Eigenschaften der Sprache, von der sie abgeleitet wurde, wodurch ein Teil des Gesamtsystems leichter handhabbar wird. Der Benutzer einer internen DSL hat den Eindruck eine Spezialsprache und nicht die Grundsprache zu benutzen.

Language Workbench Eine *Language Workbench* ist eine spezialisierte Integrierte Entwicklungsumgebung um DSLs zu definieren und zu erzeugen. Darüber hinaus stellt sie eine vom Benutzer veränderbare Umgebung dar, mit der er DSL-Skripte schreiben kann.

Nutzen von DSLs. Nach Fowler [Fow10, Chap. 2.2] hat eine DSL einen genau vorgegebenen Fokus, auf dem sie spezialisiert ist. Innerhalb eines Projektes können deswegen auch mehrerer DSLs parallel verwendet werden. Bei der Bewertung der DSLs ist es dabei wichtig zwischen dem Modell und der DSL als solche zu unterscheiden. Es gibt es vier wesentliche Vorteile von DSLs.

Verbesserte Entwicklungsproduktivität Durch eine DSL werden die Ziele eines Teils eines Systems klar definiert. Oft ist es einfacher die Definition eines Programms in Form einer DSL zu verstehen. Dadurch wird es einfacher und schneller Fehler zu finden und das System zu modifizieren. Weiterhin wird durch die beschränkte Ausdrucksstärke einer DSL es insgesamt unwahrscheinlicher einen falschen Code zu produzieren. Durch diese Faktoren wird die Entwicklungszeit verkürzt und die Softwarequalität verbessert.

Kommunikation mit Experten einer Domäne Durch eine DSL wird in bestimmten Fällen eine präzise Sprache bereitgestellt, wodurch die Kommunikation vereinfacht wird. Zumindest können die Experten einer Domäne das Systemverhalten durch den DSL-Code lesen, verstehen und ihre Verbesserungsvorschläge den Entwicklern mitteilen.

Veränderungen des Ausführungskontextes Durch eine DSL kann eine Definition eines Modells erst zur Laufzeit erfolgen. Hierdurch kann das Modell in verschiedenen Umgebungen ausgeführt werden. Weiterhin können DSLs auch genutzt werden, um ein Modell zu spezifizieren, wodurch erst anschließend der Code für die eigentliche Ausführungsumgebung generiert wird. Auch können für die DSLs die selben Versionsverwaltungen genutzt werden, die auch für den sonstigen Programmcode benutzt wird.

Alternatives Berechnungsmodell Anstatt einem imperativen Berechnungsmodell, wie es hauptsächlich in der Softwareentwicklung benutzt wird, ermöglichen DSLs ein deklaratives. Das heißt, es wird definiert, *was* berechnet werden soll, anstatt *wie* etwas berechnet werden soll. Es gibt auch andere Wege als DSLs, um das deklarative Berechnungsmodell zu benutzen. Auch hängt es von dem semantischen Modell ab, auf welchem die DSL aufbaut, ob eine imperative oder eine deklarative DSL besser ist. In vielen Fällen

können deklarative Sprachen es sehr viel leichter machen das Programm zu verändern.

Der Einsatz von DSLs bringt folglich viele Vorteile mit sich. Dies wurde in vielen Arbeiten nachgewiesen. So haben, zum Beispiel, Seipel et al. eine interne DSL in Prolog genutzt, um deklarative Expertenregeln zu repräsentieren (Vgl. [SNA17]).

2.2 Die Objektrelationale Unverträglichkeit

Objektorientierte und relationale Technologien basieren auf unterschiedlichen Paradigmen. Dadurch ergeben sich Probleme, wenn Objekte in eine Relation gespeichert werden sollen. Dies nennt man objektrelationale Unverträglichkeit (engl. *object-relational Impedance Mismatch*). Dieser Abschnitt führt diese grundsätzliche Problematik ein. Es stellt im Wesentlichen eine Zusammenfassung der Arbeit von Ireland et al. [IBNW09] dar.

Ein Paradigma ist eine bestimmte Art ein Diskursuniversum zu betrachten. Jedes Paradigma besitzt dabei seine eigene Abstraktionen, Organisationsprinzipien und Grundannahmen. Sie beeinflussen das Softwaredesign und die Softwareentwicklung einschließlich ihrer Artefakte. Dies führt zu Problemen für die Designer und Programmierer einer Anwendung, die auf verschiedene Paradigmen basiert. Eine Lösung hierfür ist typischerweise das Repräsentieren der Konzepte eines Paradigmas durch die des anderen Paradigmas.

Eine objektrelationale Anwendung ist meist größtenteils in einer objektorientierten Sprache geschrieben und benutzt relationale Datenbanken zum dauerhaften Speichern. Ein Lösungsvorschlag für die daraus resultierenden Unverträglichkeitsprobleme ist die objektrelationale Abbildung (Orm). Sie hat das Ziel, dass ein Programm in einer objektorientierten Sprache geschrieben werden kann, ohne dass der Programmierer die Abfragesprache der Datenbank, die Schemata der Datenbank noch die damit verbundene Semantik verstehen muss. Er muss sich nicht darum kümmern, *wie* ein Objekt gespeichert wird, sondern *was wann* gespeichert bzw. abgefragt werden muss. Jedoch isoliert eine Orm nicht die Datenbank von einem objektorientierten Programm. Das Design der relationalen Datenbank muss Fragen wie Datenredundanz, Datenintegrität, Datengröße, Zugangskontrolle, wechselseitiger Zugriff usw. beachten. Die Unverträglichkeitsprobleme tauchen auf, wenn dieses Design nicht mit dem Design des objektorientierten Programms übereinstimmt.

Die einzelnen Probleme sind:

Struktur-Problem Die Methoden einer Klasse definieren seine Struktur und Semantik. Eine Klasse kann auch Teil einer Klassenhierachie sein. Wie kann dies auf eine relationale Datenbank abgebildet werden?

Instanz-Problem Eine Zeile in einer relationalen Datenbank ist eine Aussage, die in einem bestimmten Diskursuniversum wahr ist. Eine Klasse kann eine beliebige Struktur haben. Welche Teile eines Objekts müssen in der Datenbank bereit gehalten werden?

Datenkapselungs-Problem Der Zustand eines Objekts kann nur von Methoden geändert werden, doch eine Zeile kann auch von einer anderen Anwendung geändert werden. Wie wird die Konsistenz zwischen dem Objekt und der Zeile sichergestellt?

Identitäts-Problem Ein Objekt hat eine Identität. Daher kann es in einem Programm oder in einer parallelen Ausführung des selben Programms weitere Objekte geben, die zwar gleich, aber nicht identisch sind. Der Primärschlüssel einer Zeile ist ein Teil seines Zustandes. Wie werden die Daten in beiden Repräsentationen eindeutig identifiziert?

Verarbeitungsmodell-Problem Objekte interagieren miteinander und der Zugriff erfolgt durch Navigation. Das relationale Modell ist deklarativ und der Zugriff erfolgt mengenbasiert. Objekte und relationale Modelle repräsentieren Referenzen in verschiedene Richtungen. Eine Transaktion benötigt auch nicht unbedingt alle Daten eines Objektes. Wie wird eine ausreichende Menge an Objekten in der Datenbank repräsentiert, gewartet und abgefragt?

Besitzer-Problem Das Klassenmodell und das relationale Schema gehört zu unterschiedlichen Entwicklerteams. Wie werden bei Veränderungen die notwendige Korrespondenz zwischen dem Klassenmodell und dem Datenbankschema gehandhabt?

Diese Probleme können durch bestimmte Strategien gelöst werden. Im Folgendem werden die Ebenen eingeführt, die die Strategien angehen müssen.

Paradigma-Ebene. Eine Orm-Strategie auf der Paradigma-Ebene muss die verschiedenen Sichtweisen der Paradigmen überbrücken. Das objektorientierte Paradigma beeinflusst das Programmdesign und das relationale Paradigma das Datenbankdesign. Hierbei gibt es keine einheitliche Terminologie und jedes Paradigma nutzt seine eigenen Bausteine, um ein Diskursuniversum zu beschreiben. Dies sind im objektorientierten Paradigma Konzepte wie Klassen, Vererbung, Objekte,

Attribute sowie Assoziationen und im relationale Paradigma Konzepte wie Relationen, Tupel und Domänen. Eine Relation entspricht einer Wahrheitsaussage in einem dazugehörigen Diskursuniversum, wohingegen ein Objekt eine beliebige Semantik besitzen kann. Ein Objekt ist eine Instanz seiner Klasse als auch aller seiner Oberklassen. Die Methoden beschreiben das Verhalten des Objekts. Ein Objekt hat eine Identität, die den aktuellen Zustand beschreibt und seine gültigen Zustände werden über Constraints definiert. Ein Tupel hat kein intrinsisches Verhalten, sondern ist Ziel eines relationalen Operators. Die Unterschiede zwischen den beiden Perspektiven führt zu der sogenannten konzeptuellen Unverträglichkeit. Eine Strategie auf der Paradigma-Ebene muss diese Unterschiede in der Perspektive, der Terminologie und der Semantik angehen.

Sprach-Ebene. Eine Orm-Strategie auf der Sprachebene muss die allgemeinen Beziehungsmuster zwischen den Datenstrukturen in einer objektorientierten Programmiersprache und den Strukturen, die in einer relationalen Sprache verfügbar sind, behandeln. Denn jede Sprache gibt das Paradigma wieder, auf dem sie basiert. Eine Klasse kann als Vorlage angesehen werden, wie ein Objekt zur Laufzeit erschaffen wird. Ein Datenbankschema ist eine Beschreibung von einer Menge von Tabellen. Eine Tabelle bezieht sich dabei auf eine Relation und eine Zeile auf einen Tupel. Eine Zeile repräsentiert Daten über eine Sache aus einem Diskursuniversum. Ein signifikanter Unterschied ist die Erweiterbarkeit der Typsysteme in objektorientierten Strukturen. Durch Klassen können neue Typen hinzugefügt werden, wohingegen es in relationalen Sprachen häufig keine gleichwertige Erweiterbarkeit gibt. Aspekte des objektorientierten Designs, wie Klasse und Objekt, müssen mittels der Konstrukte der relationalen Sprache repräsentiert werden.

Eine Spalte ist ein skalarer Wert und kann das Wesen einer Klasse inkl. seines Verhalten nicht widerspiegeln, sondern lediglich den Zustand eines Objektes speichern. Während ein Objekt zur Laufzeit eine eindeutige Identität, unabhängig von seinem Zustand, besitzt, ist der Wert des Hauptschlüssels Teil des Zustands einer Zeile. Eine Strategie auf der Sprach-Ebene beschreibt hierbei die Beziehung zwischen den Datenstrukturen, bezüglich der Struktur und der Identität. Es müssen dabei nicht nur die Unterschiede anhand der Standardisierung der Sprache berücksichtigt werden, sondern auch zwischen den konkreten Sprachversionen.

Schema-Ebene. Eine Orm-Strategie auf der Schema-Ebene erzeugt eine Abbildung zweier Repräsentationen eines Konzepts, wobei der Fokus auf Designfragen steht. Das Konzept einer objektrelationalen Anwendung wird durch mindestens zwei Schemata beschrieben. Eines, das auf Klassen, und eines, das auf Tabel-

len basiert. Der Grund rührt nicht nur von den unterschiedlichen Sprachen her, sondern auch daher, dass die Ziele eines Klassenmodells und eines relationalen Schemas unterschiedlich sind. Die Designer einer Klasse fokussieren sich auf eine geschlossene Repräsentation eines Netzwerkes interagierender Objekte. Der Fokus im relationalem Schema liegt gewöhnlich auf der Datengröße, die Datenintegrität und die Entfernung von redundanten Daten. Die daraus resultierenden Tabellen und Klassen können nicht immer eins-zu-eins übertragen werden. Eine Strategie auf der Schema-Ebene geht dies an. Sie fokussiert sich auf die Beziehung zwischen zweier unterschiedlicher Beschreibungen des selben Konzeptes und zwar im Wesentlichen mit dem Klassenmodell und dem relationalem Schema.

Als erstes Beispiel sei eine Klasse angenommen, die ein nicht-skalers Attribut, wie z. B. ein Array, besitzt. Um dies in einem Relationsschema zu speichern, kann eine eigene Tabelle angelegt werden, in der für jedes Objekt dieser Klasse und Element des nicht-skaleren Attributs eine Zeile eingefügt wird. Eine Abbildung muss wissen, dass für jedes Objekt jeder Wert seines nicht-skalaren Attributs in einer eigenen Ziele gespeichert ist. Die Informationen über das Objekt können mittels eines *Joins* über den beiden Tabellen wiederhergestellt werden.

Als zweites Beispiel sei angenommen, dass die Klasse A eine Unterklasse von B ist. Um redundante Informationen zu vermeiden, kann eine einzelne Tabelle erstellt werden, die die Klassen A und B repräsentiert und für alle Attribute aus A und B jeweils eine Spalte enthält. Eine zusätzliche Typ-Spalte gibt an, ob die Zeile ein Objekt der Klasse A oder der Klasse B repräsentiert. Eine Abbildung muss hierbei wissen, dass die Daten, die den Zustand eines Objektes der Klasse B repräsentieren, nur in den Zeilen gespeichert ist, in denen in der Typ-Spalte die Kasse B angegeben ist. Weiterhin muss sie beachten, dass die Typ-Spalte die Vererbung von A und B enthalten ist.

An diesen Beispielen lässt sich zeigen, dass durch jede zusätzliche Tabelle und Spalte ein strukturelles Problem hinzufügt wird, wodurch die Abbildung immer komplexer wird. Wenn die Daten eines Objektes auf zwei Tabellen aufgeteilt sind, muss diese Beziehung dokumentiert, kommuniziert und implementiert werden.

Instanz-Ebene. Eine wichtige Frage bei einer Orm-Strategie ist die Behandlung von Objekten, die in Struktur, Zustand und Verhalten untergliedert sind. Die Struktur eines Objektes wird sowohl über die Klasse als auch über das Datenbank-Schema definiert. Das Verhalten wird durch die Klasse definiert und ein gültiger Zustand muss sowohl im Speicher als auch über ggf. mehreren Zeilen und Tabellen gesichert werden. Die Fragmentierung ist problematisch, weil die Daten eines Objekts nicht direkt zu einer Zeile bzw. Zelle übertragen werden können. Je nach

Sprache, in der die relationale Datenbank geschrieben ist, kann das Verhalten eines Objektes gar nicht oder nur fragmentiert repräsentiert werden. Zur Laufzeit müssen nicht notwendiger Weise alle Daten eines Objektes abgefragt werden, um eine Transaktion zu vervollständigen. Es ist also wichtig, dass die Fragmentierung behandelt wird. Dies kann durch eine Abbildungsstrategie angegangen werden, die auch den Grad der Fragmentierung beeinflusst. Der gültige Zustand eines Objektes kann einerseits über Regeln in den Klassenmethoden sichergestellt werden oder anderseits durch Datenbank-Constraints. Da das Design des relationalen Schemas auch durch die Strategie auf der Schema-Ebene beeinflusst wird, haben die Entscheidungen, die bei einer Strategie getroffen werden, auch Auswirkungen auf die anderen Strategien.

Konsequenzen für die Integration von Prolog in Python

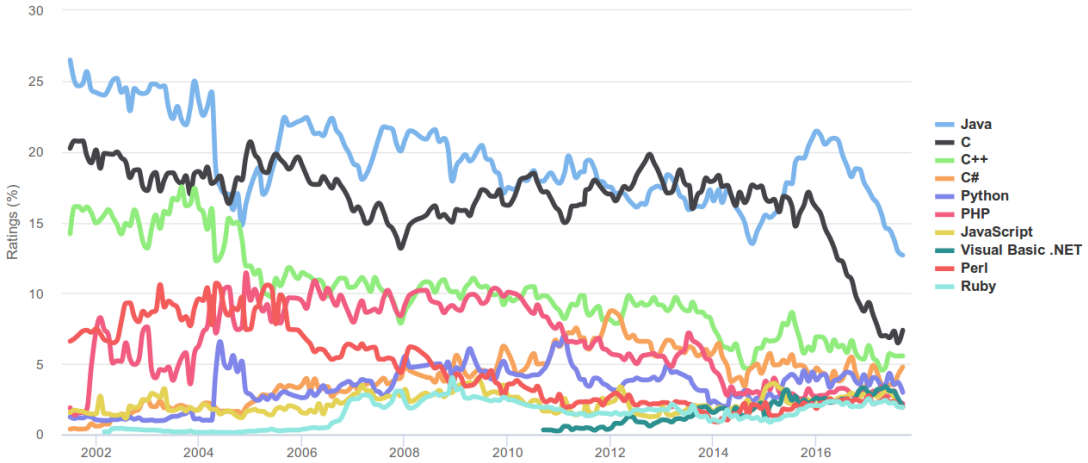
Ireland et al. zeigen auf, dass es bei der Kombination von dem objektorientierten Paradigma, wie es auch in PYTHON eingesetzt wird, und dem relationalen-Datenbank-Ansatz zu Problemen bei der Softwareentwicklung kommt. PROLOG wird normalerweise primär als logische Programmiersprache charakterisiert. Dennoch ist die Art und Weise wie Daten in einer PROLOG-Datenbank abgespeichert werden mit dem relationalem Modell vergleichbar. Daher ergeben sich zwangsläufig bei der Einbindung von PROLOG in PYTHON die vergleichbare Probleme. Vgl. die Einbindung in der Datenbankschnittstelle ODBC.

2.3 Multiparadigmen-Programmierung mit Prolog und Python

In diesem Abschnitt werden die Vorteile der Multiparadigmen-Programmierung mit PROLOG und PYTHON diskutiert. Dies betrifft die Verbreitung der Programmiersprachen einerseits und die Handhabung der objektrelationalen Unverträglichkeit anderseits.

Verbreitung der Programmiersprachen. Der TIOBE-Index bewertet monatlich die Beliebtheit von Programmiersprachen und wird von dem privaten Unternehmen *Alexa Internet* erstellt. Um den Index aufzubauen, werden die 25 populärsten Suchmaschinen ausgewählt und mit ihnen nach den Programmiersprachen gesucht. Für jede Sprache wird so die Anzahl der Suchtreffer bestimmt. Dabei wird die Konfidenz anhand der ersten hundert Einträge in einer Suchmaschine abgeschätzt,

Abbildung 2.1: Beliebtheit der Programmiersprachen. Quelle: [tiob]



um damit die Anzahl der Falsch-Negativen zu bestimmen. Diese werden von der Trefferanzahl abgezogen. Anschließend wird die Trefferanzahl noch für jede Suchmaschine normalisiert. Die Summe aus diesen Trefferanzahlen pro Suchmaschine bildet das Maß der Beliebtheit (Vgl. [tioa]). Die Abbildung 2.1 zeigt die Beliebtheit der Programmiersprachen gemäß dem Tiobe-Index von 2002 bis Anfang 2017.

Es fällt auf, das die Programmiersprache Prolog nur selten eingesetzt wird. Im Tiobe-Diagramm wird sie noch nicht einmal eingezeichnet. Bei den Durchschnittsbewertungen, die über sehr lange Zeit analysiert wurden, rangiert Prolog seit der Jahrtausendwende stets auf den hinteren Plätzen. Im Jahre 1987 war es noch die drittbeliebteste Programmiersprache.

Python war seit 2002 stets eine der zehn beliebtesten Programmiersprachen. Dennoch waren und sind Sprachen wie Java oder C beliebter [tiob].

Es gibt daher das Problem, dass Prolog, wie bereits ausgeführt, über viele Vorteile verfügt, es aber in der Masse der Programmierer kaum beachtet wird. Es ist daher wünschenswert, dass ein Weg gefunden wird, mit dem Prolog und die darauf basierenden Technologien, einer größeren Nutzerschaft verfügbar gemacht wird. Python ist eine einfach zu erlernende Programmiersprache, weil es vielen der weiter verbreiteten Sprachen ähnlich ist und eine einfache und übersichtliche Syntax besitzt. Deswegen stellt eine Integration von Prolog in Python ein wichtigen Beitrag zu diesem Ziel dar. Auch die bereits existierenden Prolog-Programme, wie ClioPatria, können so leichter verbreitet werden.

Auflösung der objektrelationalen Unverträglichkeit. Wenn die Integration von PROLOG in PYTHON so durchgeführt wird, dass ein PYTHON-Entwickler keine neuen Konzepte erlernen muss, sondern nur durch objektorientierte Strukturen PROLOG-Datenbanken verwalten kann, so wird das Problem der objektrelationalen Unverträglichkeit aus seiner Sichtweise aufgelöst.

3 Integration von Prolog in Python

Dieses Kapitel behandelt Fallstudien zu der Multiparadigmen-Programmierung in Prolog und Python. Dazu wird untersucht, in welchem Ausmaß Multiparadigmen-Programmierung mit Prolog und Python durch derzeit verfügbare Werkzeuge bereits möglich ist. Die erste Fallstudie beschäftigt sich mit der Integration von ClioPatria, einer Rdf-Bibliothek für Prolog. Die zweite Fallstudie beschäftigt sich mit der Integration des Parser ParZu, dessen Resultate auch in Prolog-kompatiblen Code ausgegeben werden können. In diesen Fallstudien werden die Problematiken der derzeit verfügbaren Werkzeugen ersichtlich. Als Ergebnis wird festgestellt, dass sie unzulänglich sind, weil sie einerseits veraltet und andererseits nur mit profunden Prolog-Kenntnissen nutzbar sind. Als Lösungsvorschlag wird die Entwicklung des neuen Integrationsframeworks CAPPy vorgeschlagen.

3.1 Bisherige Möglichkeiten zur Einbindung von Prolog in Python

Es wurden bereits verschiedene Werkzeuge für eine Einbindung von Prolog in Python entwickelt. In diesem Abschnitt wird eine Übersicht hierüber gegeben. Weiterhin wird begründet, warum lediglich PySWIP für die Fallstudien genutzt werden kann.

Anforderungen. An ein Tool, dass Prolog in Python integriert, werden in dieser Arbeit folgende Anforderung gestellt:

- Die aktuellste Version des Tools sollte vor möglichst kurzer Zeit veröffentlicht worden sein. Es werden regelmäßig neue Versionen von Python und den Prolog-Interpretern veröffentlicht, weswegen nur aktuelle Werkzeuge mit den neueren Versionen kompatibel sind.

- Das Tool soll zumindest mit dem PROLOG-Interpreter SWI-PROLOG kompatibel sein, da viele PROLOG-Programme auf SWI-PROLOG ausgelegt sind. Insbesondere ist CLIOPATRIA nur mit SWI-PROLOG kompatibel.
- Das Interface soll möglichst einfach zu bedienen sein. Das heißt, dass die Installation einfach erfolgen kann, dass die Bedingung in PYTHON benutzerfreundlich ist und es mit möglichst geringen PROLOG-Kenntnissen bedienbar ist.

Die folgende Auflistung gibt Auskunft über die existierenden Tools, die eine Anbindung von PROLOG an PYTHON bereits ermöglichen, und zeigt inwiefern sie den zuvor genannten Anforderungen entsprechen.

bedevere Dieses Werkzeug kann lediglich mit dem PROLOG-Interpreter GNU PROLOG kommunizieren. Es ist damit insbesondere nicht mit SWI-PROLOG kompatibel. Es erfüllt die zuvor genannten Anforderungen somit nicht.

PyLog (Autor: Petullo) Die Entwicklung wurde anscheinend eingestellt. Es gibt auch keine offizielle Plattform mehr, über die das Werkzeug bezogen werden kann.

PyLog (Autor: Delord) Es handelt sich hierbei um ein PYTHON-Skript, das Konstrukte bereit hält, die der logischen Programmierung angelehnt sind. Man könnte mit größerem Aufwand einen Übersetzter schreiben, der PROLOG-Programme in die Konstrukte von PYLOG überführt und so eine PYTHON-Umgebung simulieren. Es bietet keine offensichtliche Möglichkeit die Ergebnisse einer PROLOG-Abfrage unkompliziert zu verarbeiten. Es erfüllt die zuvor genannten Anforderungen somit nicht.

PySWIP Von den Entwicklern von SWI-PROLOG selbst wurde auf dieses Werkzeug hingewiesen [swi]. 2014 wurde zum letzten Mal eine neue Version von PYSWIP veröffentlicht [pys]. Es ist mit der SWI-PROLOG-Version 6 kompatibel. Damit ist PYSWIP das am ehesten auf modernen Systemen einsetzbare Werkzeug auf dieser Liste. Es ermöglicht PROLOG-Abfragen in PYTHON zu stellen, indem es sie an eine Instanz des SWI-PROLOG weiterleitet. Weiterhin stellt es einfache Klassen zur Verfügung, mit denen es einfacher ist, die Ausgaben auszuwerten. Es entspricht den zuvor genannten Anforderungen damit teilweise.

Prolog (Interface) Die Entwicklung wurde wahrscheinlich eingestellt, denn es gibt keine offizielle Plattform mehr, über die das Werkzeug bezogen werden kann.

Listing 3.1: Starten von ClioPatria in Prolog

```
1 use_module('/path/to/cliopatria').
2 cp_server.
```

pwig Die neuste Version dieses Werkzeugs stammt aus dem Jahr 2004. Es ließ sich trotz größeren Aufwands auf den verschiedenen Testrechnern nicht mehr kompilieren. Es ist daher nicht mehr mit moderner Software ausführbar, insbesondere auf keinem modernen Betriebssystem mit den aktuellen Versionen von Swi-Prolog und Python. Es erfüllt die zuvor genannten Anforderungen somit nicht.

PySWIP ist das modernste Werkzeug auf dieser Liste. Es ist noch bezugsfähig, auf einem modernen System ausführbar und ermöglicht eine direkte Anbindung an Swi-Prolog. Die folgenden Fallstudien wurden daher mit diesem Werkzeug durchgeführt.

3.2 Fallstudie: Einbindung der Bibliothek ClioPatria in Python

In diesem Abschnitt wird eine Fallstudie durchgeführt, in der die Einbindung der Rdf-Bibliothek ClioPatria in Python untersucht wird. Dazu wird zunächst die Funktionsweise von ClioPatria in Swi-Prolog erläutert und anschließend eine Einbindung in Python mittels PySWIP durchgeführt.

Funktionsweise von ClioPatria am Beispiel von Yago. Um ClioPatria in Swi-Prolog zu nutzen, muss die ClioPatria-Bibliothek geladen und der Server gestartet werden, siehe Listing 3.1. Im Anschluss daran ist ClioPatria einsatzfähig. Es können, wie im Listing 3.2 gezeigt, Rdf-Dateien geladen werden (Zeile 1), Rdf-Tripel gesucht werden (Zeile 2) oder eine Sparql-Abfrage gestartet werden (Zeile 3). Die Rdf-Tripel aus Yago werden wie gewöhnliche Rdf-Tripel in ClioPatria dargestellt. So wird der Rdf-Tripel, der besagt, dass die Kategorie der mazedonischen *New-Wave*-Musikgruppen eine Unterklasse der Klasse der musikalischen Organisationen ist, wie in Listing 3.3 in ClioPatria repräsentiert. Die Abfrage, wer an der Universität Würzburg arbeitet und einen Nobelpreis in Physik erhalten hat, kann wie in Listing 3.4 als Abfrage formuliert werden. Die Abfrage kann von einem Swi-Prolog-Interpreter ausgewertet werden, siehe Listing 3.5.

Listing 3.2: Verwendung von ClioPatria in Prolog

```
rdf_db:rdf_load(+FileName).
rdf_db:rdf(?X, ?Y, ?Z).
sparql:sparql_query(+Query, -Result,[]).
```

Listing 3.3: Rdf-Tripel aus Yago in ClioPatria

```
('http://yago-knowledge.org/resource/
 ↪ wikicat_Macedonian_New_Wave_musical_groups',
 ↪ rdfs:subClassOf,
 ↪ 'http://yago-knowledge.org/resource/
 ↪ wordnet_musical_organization_108246613').
```

Listing 3.4: Abfrage an Yago durch Prolog-Konstrukte

```
(X, 'http://yago-knowledge.org/resource/hasWonPrize',
 ↪ 'http://yago-knowledge.org/resource/
 ↪ Nobel_Prize_in_Physics'),
 ↪ (X,'http://yago-knowledge.org/resource/worksAt',
 ↪ 'http://yago-knowledge.org/resource/
 ↪ University_of_Würzburg').
```

Listing 3.5: Auswertung der Abfrage an Yago (siehe Listing 3.4)

```
X = 'http://yago-knowledge.org/resource/
 ↪ Wilhelm_Röntgen'.
```

Listing 3.6: Prolog-Hilfsdatei für das Ansprechen von ClioPatria

```
1 :- use_module('../ClioPatria/cliopatria').
2 :- cp_server.
3 load_rdf_file(FileName) :- rdf_db:rdf_load(FileName).
4 find_rdf_triple(X,Y,Z) :- rdf_db:rdf(X,Y,Z).
```

Listing 3.7: Laden der Yago-Tripel in Python

```
1 from pyswip import *
2 prolog = Prolog()
3 self.prolog.consult("helperFile.pl")
4 load_rdf_file = Functor("load_rdf_file", 1)
5 call(load_rdf_file('path/to/yagoRdfFile'))
```

Abfragen an ClioPatria in Python. Damit in Python Abfragen an Yago über ClioPatria gestellt werden können, bedarf es eines Integrationswerkzeuges. In diesem Abschnitt wird gezeigt, wie mittels PySWIP ClioPatria gestartet werden kann und die Abfrage aus dem Listing 3.4 in Python ausgewertet wird.

Da in PySWIP keine einfachen Abfragen ohne Funktoren oder Variablen möglich sind und es keine Unterstützung für Modulnamen bietet, bedarf es zunächst einer Prolog-Hilfsdatei (siehe Listing 3.6), die zuerst manuell erstellt und anschließend konsultiert werden muss. Anschließend kann ClioPatria, wie in Listing 3.7, angesprochen werden. In Zeile 1 wird das PySWIP-Modul importiert. In Zeile 2 wird eine Instanz von PySWIP erzeugt und in der folgenden Zeile die Hilfsdatei konsultiert. Dabei wird ClioPatria in einer Swi-Prolog-Instanz geladen und der Server gestartet. Mit den Zeilen 4 und 5 wird die Rdf-Datei geladen, die die Yago-Tripel enthält.

Danach kann die vorherige Abfrage an Yago wie in Listing 3.8 durch eine Methode in Python ausgewertet und auf der Standartausgabe ausgegeben werden. In der Zeile 2 wird der Funktor `find_rdf_tripel` aus der Prolog-Hilfsdatei deklariert. In Zeile 3 wird `X` als eine Prolog-Variable definiert. In den Zeilen 4 bis 11 wird eine Abfrage `q` erzeugt. In den Zeilen 12 bis 13 wird über die Ergebnistupel aus der Abfrage `q` iteriert und die Ergebnisse auf der Standart-Ausgabe ausgegeben. In Zeile 14 wird die Abfrage `q` wieder geschlossen. Es fällt auf, dass man für die Benutzung von PySWIP gute Prolog-Kenntnisse benötigt. Auch müssen alle Funktoren und Variablen vorab definiert werden.

Listing 3.8: Yago-Abfrage in Python (Vgl. Listing 3.4)

```
def printRDFTriples(self):
  find_rdf_triple = Functor("find_rdf_triple", 3)
  X = Variable()
  q = Query(find_rdf_triple(X,
   ↪ 'http://yago-knowledge.org/resource/hasWonPrize',
   ↪ 'http://yago-knowledge.org/resource/
   ↪ Nobel_Prize_in_Physics'),
   ↪ find_rdf_triple(X,
   ↪ 'http://yago-knowledge.org/resource/worksAt',
   ↪ 'http://yago-knowledge.org/resource/
   ↪ University_of_Würzburg')
  while q.nextSolution():
    print X.value
  q.closeQuery()
```

3.3 Fallstudie: Verarbeitung natürlicher Sprache

In diesem Abschnitt wird zunächst der Parser ParZu eingeführt. Anschließend werden Möglichkeiten diskutiert, wie seine Resultate durch eine Kombination von Prolog und Python sinnvoll verarbeitet werden können und eine solche Verarbeitung mittels PySWIP vorgestellt.

3.3.1 Der Parser ParZu

Der ParZu ist ein Parser für die deutsche Sprache und wurde von Sennrich et al. entwickelt. Seine Ursprünge liegen beim Parser Pro3Gres [SHM08], der für die englische Sprache entwickelt wurde. Deshalb gibt es einige Gemeinsamkeiten zwischen den beiden Architekturen. Dieser Abschnitt führt den ParZu ein und ist, soweit nicht anders gekennzeichnet, eine Zusammenfassung der Arbeit von Sennrich et al. (Vgl. [SVS13]).

Vereinfacht gesagt wandelt der ParZu einen Text in deutscher Sprache in eine auf die Dependenzgrammatik angelehnten Form um, die von Maschinen leicht verarbeitet werden kann. Nach Tarvainen [Tar00, S. IX] "[untersucht] die Dependenzgrammatik (...) Abhängigkeitsbeziehungen zwischen Satzteilen. Sie will ermitteln, welche regierenden Elemente verschiedenen Ranges es im Satz gibt und was für

untergeordnete Elemente mit ihnen verbunden sind. Außerdem versucht die Dependenzgrammatik, die verschieden Satzteile mit Hilfe von linguistischen Operationen explizit und möglichst widerspruchslos zu definieren". Er steht damit im Gegensatz zu den Parsern, die die ebenfalls weitverbreitete Phrasenstrukturgrammatik anwenden. Nach Chomsky [Cho56] wird bei einer Phrasenstrukturgrammatik ein Satz in verschiedenen Phrasen aufgeteilt, die wiederum immer weiter in konstituierende Phrasen aufgeteilt werden, bis die endgültigen Bestandteile erreicht werden. Diese Aufteilung dient der syntaktischen Beschreibung des Satzes.

Die wesentlichen Komponenten der PARZU-Architektur sind:

- Eine handgeschriebene Grammatik, die weitgehend unabhängig von konkreten Wörtern ist und in erster Linie dem Part-of-Speech-Tagging dient. Part of Speech (POS) werden nach Martin und Jurafsky [MJ99, S. 285] äquivalente Klassen genannt, in denen Wörter traditionell gruppiert werden. In klassischen Grammatiken gibt es nur wenige solcher Klassen (z. B. Substantive, Verben, Adjektive), während es in moderne Grammatiken wesentlich mehr gibt
- Externen Anwendungen für das POS-Tagging und den morphologischen Analysen
- Einem statistischem Modul, das sprachliche Mehrdeutigkeiten auflöst

Eine Baumbank ist ein Text-Korpus, bei dem die darin beinhalteten Sätze vorab geparst wurden. Der PARZU wurde auf der Baumbank *TüBa-D/Z* trainiert, die rund 65.500 Sätze aus einer deutschen Zeitung enthält und die von Hand annotiert wurden.

Zunächst wurde eine Version von PARZU veröffentlicht, die verschiedene Werkzeuge mit restriktiveren Softwarelizenzen genutzt hatte [SSVW09]. In einer neuen Version wurden die Komponenten zum Großteil durch solche, mit einer freieren Lizenz ausgetauscht. Weiterhin wurden einige Performanceverbesserungen eingearbeitet. Das POS-Tagging wurde verbessert, indem morphologische Informationen miteinbezogen wurden. Die Auswahl der besten POS-Sequenz wurde anhand von syntaktischen Informationen verbessert. Eine weitere Performancesteigerung wurde erreicht, in dem bereits geparste Texte als Trainingsdaten für das POS-Tagging und dem statistischen Parsen eingesetzt wurden.

Morphologie-Werkzeuge. In der neuen Version werden die Morphologie-Werkzeuge SMOR bzw. MORPHISTO eingesetzt. Beim Parsen stellen solche Tools zwei Arten von nützlichen Informationen bereit:

- Informationen über die Grundformen eines Wortes sorgen für eine weniger dichte Repräsentation der statistischen Daten
- Analysen über die Flexionen (d. h. grammatikalisch bedingte Änderungen eines Wortes innerhalb eines gegebenen Satzes) vereinfachen die Auflösungen von Mehrdeutigkeiten in den Nominalphrasen. Weiterhin helfen sie, Bedingungen zu formulieren, mit denen sichergestellt wird, dass bestimmte Eigenschaften verschiedener Wörter übereinstimmen, der sogenannten Kongruenz. So ist z. B. ist der Satz *Der interessanten Vorlesung wird viel besucht* hinsichtlich der Kongruenz inkorrekt. Da das Wort *Vorlesung* feminin ist und in dem Satz im Singular steht, muss auch der vorangehende Artikel und das Adjektiv dieser Form entsprechen.

In der Evaluation haben sowohl die neuen als auch die bisherigen Morphologie-Werkzeuge gute Ergebnisse geliefert. Die Verbesserung durch die morphologische Analyse ist insgesamt signifikant, dennoch liefert PARZU auch ohne sie bereits gute Ergebnisse.

POS-Tagging-Werkzeuge. Das Morphologie-Werkzeug extrahiert die möglichen POS-Tags, indem es alle Analysen der Wortform auf eine standardisierte Tag-Menge abbildet. Doch nicht alle Wortformen können anhand der lokalen Umgebung erkannt werden, obwohl die morphologischen Analyse dies bereits erleichtert. Ein Beispiel hierfür ist, dass sich nicht immer ohne Weiteres entscheiden lässt, ob ein Verb in der finiten oder infiniten Form steht. Sollte beim PARZU ein solcher Fall auftreten, so weist er im Zweifelsfall keinen Tag zu. Dadurch wird verhindert, dass ein falscher Tag zugewiesen wird und sorgt damit für eine höhere Präzision. PARZU bietet aber auch die Möglichkeit die Menge der n-besten Tags zu ermitteln, um anschließend in einer vollständigen Analyse mithilfe diverser, aus dem PARZU extrahierten syntaktischen Features, die beste Tag-Sequenz zu ermitteln. Ein solches n-Beste-Tagging vermeidet deutlich viele Tagging-Fehler. Hierzu setzt PARZU sogenannte Conditional-Random-Field-Tagger (CRF-Tagger) ein, um die n-besten Tags zu bestimmen. Konkret setzt der PARZU WAPITI als CRF-Tagger für das Training und für die Dekodierung ein. Im Vergleich mit dem ursprünglichen Ansatz liefert WAPITI vergleichbare Ergebnisse. In Kombination mit einem Morphologie-Werkzeug liefert es geringfügig bessere Ergebnisse.

Gesamt-Evaluation. Im Vergleich mit einem anderem aktuellen Parser, dem MALTOPTIMIZER, schneidet die ältere Version vom PARZU schlechter ab, wohingegen die neue Version vom PARZU ihn sogar leicht übertrifft. Dies liegt vor allem an den weiterentwickelten Kernkomponenten, also der Grammatik und dem Di-

sambiguierungsmodul (einem Modul, das Mehrdeutigkeiten auflöst). Ein wichtiger Unterschied machen hierbei die Daten, auf denen der Parser trainiert wurde. Nur bei Trainingsdaten, die dem Gold-Standard entsprechen, konnte die höhere Leistung erreicht werden. Ansonsten liegt sie leicht darunter. Dies liegt daran, dass der ParZu dazu tendiert im Zweifelsfall lieber keinen Tag zuzuweisen, anstatt einen falschen. Bei dem Vergleich wurde weiterhin nur die Güte der Ergebnisse verglichen, nicht die Zeit, die die Parser gebraucht haben. Die Parser haben unterschiedliche Stärken und Schwächen, wenn man bestimmte Teilprobleme betrachtet. Das heißt, je nach Art des Textes könnte die Leistung auch anders ausfallen.

3.3.2 Verarbeitung natürlicher Sprache in Prolog

Wird in einem PYTHON-Programm eine Komponente für die Verarbeitung natürlicher Sprache benötigt, so bietet sich eine Integration von PROLOG an, wegen seiner guten und nativen Unterstützung für die Verarbeitung natürlicher Sprache. In diesem Abschnitt werden Möglichkeiten aufgezeigt, wie die Parsing-Ergebnisse des ParZus mittels PROLOG in PYTHON verarbeitet werden können. Als erstes wird gezeigt, wie ein Baum erzeugt werden kann, der die Abhängigkeiten der Terme eines natürlich-sprachlichen Textes repräsentiert. Weiterhin wird skizziert, wie eine Verarbeitung der Parsing-Ergebnisse durch eine Integration von PROLOG und PYTHON mittels PySWIP möglich ist. Im Anschluss wird gezeigt, welches Potential durch eine Integration von ParZu in PYTHON entsteht.

Erzeugen eines Baumes, der die Abhängigkeiten der Terme wiedergibt. In diesem Abschnitt wird gezeigt, wie ein Baum erzeugt werden kann, der die Abhängigkeiten der Terme eines natürlich-sprachlichen Textes repräsentiert. Hierzu wird zunächst dem Parser ParZu ein Text zur Verarbeitung übergeben. Das Parsing-Ergebnis kann mittels PROLOG-Abfragen einfach verarbeitet werden. Hierzu ist es nötig, dass von PYTHON aus auf einen PROLOG-Interpreter zugegriffen werden kann. Von den existierenden Werkzeugen bietet sich hierfür PySWIP an.

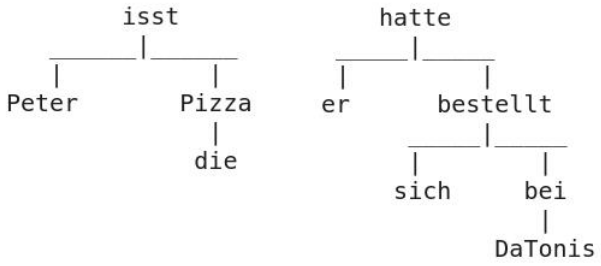
Betrachten wir im Folgenden den Beispielsatz: "Peter isst die Pizza, die er sich bei DaTonis bestellt hatte". Setzt man in ParZu das Ausgabeformat auf PROLOG, so liefert das Programm ein Ergebnis, das in Listing 3.9 eingesehen werden kann.

Im Folgenden wird immer davon ausgegangen, dass nur Wörter des selben Satzes (`Sentence`-Komponente) betrachtet werden. Die Komponente `HeadPosition` gibt an, welches Wort der Kopf des Faktes ist. Da es sich um eine Dependenzgrammatik handelt, ist der Kopf immer ein Verb, es sei denn, das Wort ist bereits selbst der

Listing 3.9: Parsing-Ergbenis in Prolog-Schreibweise

```
% word(Sentence, Position, WordForm, Lemma, POS, DepRel,
% HeadPosition, Morphology).
word(1, 2, isst, essen, 'VVFIN', root, 0,
 ↪ [['3', 'Sg', 'Pres', 'Ind']]).
word(1, 3, die, die, 'ART', det, 4,
 ↪ [['Def', 'Fem', 'Nom', 'Sg'],
 ↪ ['Def', 'Fem', 'Acc', 'Sg']]).
word(1, 4, 'Pizza', 'Pizza', 'NN', obja, 2,
 ↪ [['Fem',' Nom', 'Sg'],
 ↪ ['Fem', 'Acc', 'Sg']]).
word(1, 5, ',', ',', '\$,', root, 0, \_G3183).
word(1, 6, die, die, 'PRELS', root, 0,
 ↪ [['Fem', 'Nom', 'Sg'], ['Fem', 'Acc', 'Sg'],
 ↪ [\_G3201, 'Nom', 'Pl'], [\_G3213, 'Acc', 'Pl']]).
word(1, 7, er, er, 'PPER', subj, 12,
 ↪ [['3', 'Sg', 'Masc', 'Nom']]).
word(1, 8, sich, sie, 'PRF', obja, 11,
 ↪ [['3', 'Sg', 'Dat'], ['3', 'Sg', 'Acc'],
 ↪ ['3', 'Pl', 'Dat'], ['3', 'Pl', 'Acc']]).
word(1, 9, bei, bei, 'APPR', pp, 11, [['Dat']]).
word(1, 10, 'DaTonis', 'DaTonis', 'NE', pn, 9,
 ↪ [[\_G3177, 'Dat', \_G3183]]).
word(1, 11, bestellt, bestellen, 'VVPP', aux,
 ↪ 12, \_G3183).
word(1, 12, hatte, haben, 'VAFIN', root, 0,
 ↪ [['3', 'Sg', 'Past', 'Ind']]).
word(1, 13, '.', '.', \$., root, 0, \_G3183).
word(1, 1, 'Peter', 'Peter', 'NE', subj, 2,
 ↪ [[\_G3177, 'Nom', 'Sg']]).
```

Abbildung 3.1: Abhängigkeiten der Terme in Baumdarstellung



Listing 3.10: Prolog-Abfrage für den Kopf einer Dependenz

```
1 word(_, _,Orginal_Form,Lex_Form,_,root,_,_).
```

Kopf der Regel (der Eintrag ist in diesem Falle 0). Durch diese Fakten lässt sich von Hand der Baum in Abbildung 3.1 erstellen.

Das finden der Abhängigkeiten lässt sich elegant durch Prolog-Regeln automatisieren. Diese können durch einen Prolog-Interpreter ausgewertet werden und die Informationen über einem Gateway einem Python-Programm zur Verfügung gestellt werden. Sei zum Beispiel angenommen, dass aus dem Parsing-Ergebnis in Listing 3.9 die Wörter mitsamt ihrer Grundform abgefragt werden sollen, die den Kopf einer Dependenz bilden. In Prolog kann dies durch eine kurz Abfrage formuliert werden, siehe Listing 3.10 Die Ergebnisse dieser Abfrage sind zur Übersichtlichkeit in der Tabelle 3.1 aufgelistet. Es fällt auf, dass die Terme `"."`, `"die"` und `","`nicht mit den anderen Satzbausteinen verknüpft werden konnten. Bei den Satzzeichen ist dies nachvollziehbar. Dass das Wort `"die"` einen Nebensatz einleitet, der das Wort `"Pizza"`genauer beschreibt, scheint der ParZu nicht zu erkennen. Soll nun eine solche Abfrage in Python mit PySWIP formuliert werden,

Tabelle 3.1: Ergebnis der Abfrage in Tabellenform (siehe Listing 3.10)

Org_Form	**Lex_Form**
isst	essen
die	die
hatte	haben
’,’	’,’
’.’	’.’

Listing 3.11: Bestimme Kopf einer Dependenz in Python (Vgl. Listing 3.10)

```
1 word = Functor("word", 8)
2 OrgForm = Variable()
3 LexForm= Variable()
4 _ = Variable()
5 q = Query(word(_, _,  Orginal_Form, LexForm, _, "root",
6  ↪ _, _))
7 while q.nextSolution():
8    print X.value
9 q.closeQuery()
```

so kann dies wie in Listing 3.11 geschehen. In Zeile 1 wird zunächst `word` als ein Funktor mit acht Stellen deklariert. In den Zeilen 2 bis 4 werden die Variablen für die Abfrage deklariert. Zu beachten ist hier, dass selbst die anonyme Variable `"_"` explizit deklariert werden muss. In den Zeilen $5 - 6$ wird die Abfrage `q` erstellt. Auf die Ergebnisse der Abfrage können wie in den Zeilen 7 und 8 über die Variable `q` zugegriffen werden.

Weitere Möglichkeiten das Parsing-Resultat des ParZus zu verwerten. Der ParZu liefert bereits zu einer Eingabesequenz ein *Part-of-Speech-Tagging* als ein Teil seiner Ausgabe. Hierauf aufbauend können verschiedene Softwareanwendungen in Python unterstützt werden. Nach Jurafsky und Martin [MJ99, S. 285-286] gibt es unterschiedliche Einsatzmöglichkeiten für das POS-Tagging:

- Das POS-Tagging für ein Wort liefert bestimmte Informationen über das Wort und seine Nachbarn. Hierbei gibt es gröbere und feinere Unterschiede zwischen den Wortarten, je nachdem wie die Tag-Menge aufgebaut ist. Ist die Wortart eines Wortes bekannt, so kann man abschätzen, welche Wörter in seiner Umgebung wahrscheinlich auftauchen werden. Dies kann für Spracherkennung genutzt werden
- Die Wortart kann bei der Entscheidung helfen, wie ein Wort ausgesprochen wird. Das kann Sprachsynthese-Systeme verbessern
- Das POS-Tagging kann genuzt werden, um das Stemming in Information-Retrieval-Systemen zu verbessern, weil es dadurch leichter wird zu erkennen, welche morphologischen Affixe ein Wort haben kann

- Auch kann es ein Information-Retrieval-System dahingehend unterstützen, dass es leichter wichtige Wörter aus einem Dokument erkennen kann (beispielsweise können Nomen wichtiger sein als Präpositionen)
- Das POS-Tagging unterstützt auch Algorithmen für das automatische Auflösen eines mehrdeutigen Wortsinns, sowie höhere Sprachmodelle für automatisierte Spracherkennung
- Sehr oft wird POS-Tagging auch für teilweises Parsen von Texten genutzt. Zum Beispiel, um schnell Namen oder andere Phrasen für die Informationsextraktion zu finden
- Mit POS-Tags versehene Korpora können weiterhin in der linguistischen Forschung eingesetzt werden. Zum Beispiel, um Instanzen oder Häufigkeiten von bestimmten Konstruktionen in großen Korpora zu finden

POS-Tagging wird also vor allem als Grundlage für aufwendigere Sprachverarbeitungen eingesetzt. Eine andere Einsatzmöglichkeit wäre es, einen erweiterten Parser zu erschaffen, der mithilfe dieser Informationen bessere Ergebnisse liefert. Als Grundlage für einen weiteren Parser bieten sich folglich zwei Möglichkeiten an:

1. Korpora mit POS-Tags können von Parsern genutzt werden, um ihre Modelle besser trainieren zu können. Der ParZu nutzt bereits einen solchen Mechanismus.
2. Die Resultate von einem vom ParZu geparsten Text können zum verfeinertem Parsen größere Texte genutzt werden. Zum Beispiel, um bestimmte Informationen, wie z. B. Namen, schneller zu finden.

Durch die Integration von PROLOG und PYTHON stehen einem solchen, erweiterten Parser viele Entwicklungsmöglichkeiten offen, Technologien aus dem PROLOG-Bereich und dem PYTHON-Bereich zu kombinieren.

3.4 Resultierende Probleme

In den vorangegangen Abschnitten wurde anhand von Fallstudien gezeigt, dass die Multiparadigmen-Programmierung mit den Programmiersprachen PROLOG und PYTHON über ein großes Potential verfügt. Es existieren bereits Werkzeuge, die dies erleichtern sollen. Allerdings kristallisierten sich bei den Fallstudien einige Probleme heraus, die im Folgenden dargelegt werden.

3.4.1 Allgemeine Probleme der Werkzeuge

Die Recherche nach möglichen Werkzeugen für die Multiparadigmen-Programmierung in PROLOG und PYTHON (siehe Abschnitt 3.1) hat ergeben, dass es zwar einige Werkzeuge gibt, mit denen dies realisiert werden könnte, allerdings wurde auch ersichtlich, dass die meisten nur sehr beschränkt in einer modernen Umgebung eingesetzt werden können, was im Folgendem begründet wird.

Veraltete Software. Ein wichtiger Nachteil der bereits existierenden Tools ist, dass sie zum größten Teil veraltet sind. Die letzte Veröffentlichung der Tools reicht von drei bis über zehn Jahre vor Erstellung dieser Arbeit zurück. Dadurch ist ihre Produktivität in modernen Umgebungen eingeschränkt bis nicht gegeben. Der Grund hierfür ist, dass stets neue Versionen der PROLOG-Interpreter und von PYTHON entwickelt und veröffentlicht werden. Dadurch ändern sich auch immer wesentliche Spracheigenschaften wie z. B. die Syntax. Weil die Tools auf diese Neuerungen nicht mehr angepasst werden, ergibt sich das Problem der Inkompatibilität.

Weiterhin bieten die Entwickler teilweise keine offene Bezugsmöglichkeit für die Tools mehr an. Daher konnte die Funktionalität mancher Tools nicht mehr evaluiert werden. In anderen Fällen konnten die Tools nicht mehr kompiliert werden, da sich einige Komponenten der Betriebssysteme im Laufe der Zeit geändert haben.

Die wünschenswerte Eigenschaft, dass die Tools in einer modernen Umgebung und insbesondere mit den aktuellen Versionen von PROLOG und PYTHON einsetzbar sind, ist größtenteils nicht gegeben.

Beschränkungen auf bestimmte Interpreter. Die bisherigen Werkzeuge beschränken sich auf einen bestimmten Interpreter von PROLOG (oft SWI-PROLOG). Das bedeutet eine signifikante Einschränkung für den PYTHON-Entwickler, denn er kann den PROLOG-Interpreter nicht mehr flexibel aussuchen. Er muss gezwungenermaßen das Werkzeug wählen, dass das Integrationswerkzeug vorschreibt. Die bereits zuvor beschriebene stark begrenzte Auswahl an Tools macht es ihm gegebenenfalls unmöglich ein Tool zu finden, dass mit seinem gewünschten PROLOG-Interpreter kompatibel ist.

Soll während der Entwicklung des Projekts der zu nutzende PROLOG-Interpreter ausgewechselt werden, so kann das gegebenenfalls bedeuten, dass der komplette Code, der sich auf das Tool bezieht, ausgetauscht werden muss. Je nach nachdem in welchem Umfang PROLOG-Interaktionen in dem Projekt verwendet werden, müssen dann große Codeteile umgeschrieben werden.

Art und Weise der Interaktion. Alle Tools haben gemein, dass sie eine aufwendige Einarbeitung für einen PYTHON-Entwickler bedürfen, sofern sie keine PROLOG-Kenntnisse besitzen. Die Tools etablieren zwar eine Verbindung zu einem PROLOG-Interpreter und stellen so eine Möglichkeit bereit, mit nativer Syntax Abfragen zu stellen, jedoch ist die Semantik der Interfaces stets sehr stark an die Programmiersprache PROLOG angelehnt. Eine Einarbeitung in die Tools beinhaltet damit indirekt, dass der Programmierer auch PROLOG erlernen muss. Der Aufwand dafür ist nicht nur unerheblich, weil unter anderem PYTHON und PROLOG andere Programmierparadigmen verwenden. Der PYTHON-Programmierer kann trotz der PYTHON-nativen Syntax nicht auf der PYTHON-typischen Art und Weise Abfragen stellen. Auch dies reduziert die Usability erheblich.

3.4.2 Probleme des Prolog-Werkzeugs PySWIP

Im diesem Abschnitt wird auf die konkreten Probleme des Werkzeugs PYSWIP eingegangen, das sich als einzige Möglichkeit zur Durchführung der Fallstudien erwiesen hat.

Zunächst lässt sich feststellen, dass dieses Werkzeug zuletzt im Jahre 2014 aktualisiert wurde. Aufgrund der über dreijährigen Zeitspanne bis zur Veröffentlichung dieser Arbeit, liegt der Vermutung nahe, dass die Entwicklung eingestellt wurde. Es benötigt nach Entwickler-Angaben eine Python-Version von mindestens 2.3 und eine SWI-PROLOG-Version von mindestens 5.6.x. Im Test konnte das Tool nur mit Python-Versionen bis höchstens 2.7 und einer SWI-PROLOG-Version unter 6.4.x zum Laufen gebracht werden. Den Anspruch, in einer Software-Umgebung auf dem neusten Stand der Technik lauffähig zu sein und die Interoperabilität für verschiedene PROLOG-Implementierungen zu wahren, wird PYSWIP nicht gerecht.

Installation. Bereits bei der Installation von PYSWIP zeigen sich größere Mängel hinsichtlich der Usability. Bereits die Notwendigkeit eine alte SWI-PROLOG-Version zu verwenden, widerspricht den Anspruch vieler Nutzer, stets mit der aktuellen Version zu arbeiten. Damit beide Versionen parallel auf einem System installiert werden können, müssen hierfür zusätzliche Vorkehrungen getroffen werden.

Weiterhin muss das SWI-PROLOG unter Umständen selbst kompiliert werden. Und zwar genau dann, wenn

Listing 3.12: Laden von PySWIP mittels des `export`-Befehls

```
1 export LD_PRELOAD=/path/to/swiprolog/library/libswipl.so
```

- Die Installationsroutine es nicht als dynamische Bibliothek installiert hat (Der Compiler muss mit dem Parameter `-enable-shared` aufgerufen werden) oder
- ein 64-Bit-Rechner genutzt werden soll (setze `CFLAGS -ggdb`)

Erschwerend kommt hinzu, dass nicht das Installationsskript von Swi-Prolog genutzt werden kann, das sowohl die Kernkomponente als auch alle Pakete installiert. Vielmehr müssen die Installationsskripte für die Kernkomponente als auch für jedes gewünschte Paket separat aufgerufen werden. Gerade für Nutzer, die wenig Erfahrung im Kompilieren haben, stellt dies eine nicht zu vernachlässigende Hürde dar. Für kundige Nutzer stellt dies zumindest einen unnötigen Mehraufwand dar.

Benutzung. In den Fallstudien stellte sich eine erschwerte Benutzbarkeit von PySWIP heraus. Insbesondere stellen die Entwickler weder Anleitung noch Referenzverzeichnis bereit. Es stehen lediglich eine kleine Auswahl an Beispielen bereit, bei denen die Anwendung von PySWIP demonstriert wird. Einige dieser Beispiele konnten auf keinem Testrechner ausgeführt werden, vermutlich weil sie Funktionalitäten nutzen, die nur in vorherigen Versionen von PySWIP zur Verfügung standen. Das Erlernen der Bedienung des Werkzeugs stellt damit für Entwickler eine weitere Hürde dar.

Als besonders schwerer Mangel lässt sich konstatieren, dass das PySWIP-Interface sehr stark an Prolog angelehnt ist. Sie steht damit dem Python-typischen Programmierstil diametral entgegen. Es lässt sich die These aufstellen, dass man PySWIP nur nutzen kann, wenn man zunächst die Grundlagen von Prolog erlernt hat.

Als weiteres Problem wurde deutlich, dass auf einigen Testrechner PySWIP die Prolog-Bibliothek nicht laden kann. Da keinerlei Möglichkeiten besteht, das Tool zu konfigurieren (ohne den Quellcode zu bearbeitet), kann der Pfad zur Prolog-Bibliothek auch nicht manuell angeben werden. Der Nutzer von PySWIP muss in diesem Fall selbst die Bibliothek laden. Eine Möglichkeit auf Linux-Rechnern hierfür ist, vor jedem Aufruf des Python-Programms mittels dem Kommando `export` die Bibliothek zu laden, wie zum Beispiel im Listing 3.12.

3.5 Ein neues Integrationsramework für Prolog in Python

Die zuvor aufgezeigten Hindernisse zeigen auf, dass mit derzeitigen Mitteln eine vollständige Integration von ClioPatria in Python schwer möglich ist. Ein Beispiel dafür ist, dass in PySWIP die Abfragen an Prolog vorprogrammiert werden müssen. Allerdings muss hierbei von einer festen Stelligkeit ausgegangen werden, was bei manchen Abfragen nicht möglich ist. Um dies zu umgehen könnte eine Prolog-Abfrage von einem neu zu schreibenden Werkzeug analysiert werden, das einen entsprechenden Code für jede Abfrage erzeugt, mit dem durch PySWIP die Abfrage ausgewertet werden kann. Der Aufwand dafür kommt der Neuentwicklung einer neuen Anbindung von Prolog und Python gleich.

Durch die Fallstudien wird insgesamt deutlich, dass es einen Bedarf für ein Werkzeug gibt, das insbesondere

- mit aktuellen Versionen von Prolog und Python in einer aktuellen Umgebung einsetzbar ist (wobei idealerweise auch die Interoperabilität für verschiedene Prolog-Interpreter gewahrt bleiben soll)
- und weiterhin Prolog für Python-Entwickler auf eine Python-typische Weise zugänglich gemacht wird

In den weiteren Kapitel werden die dafür notwendigen Funktionalitäten eines solchen Werkzeugs analysiert, die Machbarkeit evaluiert und anschließend implementiert.

4 Entwurf des Integrationsframeworks CAPPy

In diesem Kapitel wird die Machbarkeit eines neuen Integrationsframeworks für Python und Prolog evaluiert und diskutiert. Anhand dieser Erkenntnisse wird gezeigt, dass ein Werkzeug für die Einbindung von Prolog in Python möglich ist, indem ein Werkzeug für die Integration von Java und Prolog als Vorlage benutzt wird. Dieses neue Integrationsframework wird im Folgenden CAPPy (Connector Architecture for Prolog and Python) genannt.

Es wird zunächst CAPJa (Connector Architecture for Prolog and Java) eingeführt. Dies ist ein Integrationsframework für Java und Prolog. Da Java und Python viele Gemeinsamkeiten haben, kann eine Architektur wie z. B. CAPJa als Vorlage für das neue Werkzeug genutzt werden. Für die Evaluation werden die wesentlichen Konzepte von CAPJa auf abstrakter Ebene herausgearbeitet.

Darauf aufbauend wird geprüft, inwiefern diese Konzepte auf ein neues Integrationsframework für Python und Prolog übertragbar sind und welche Probleme es sowohl auf abstrakter als auch auf implementierungsspezifischer Ebene sich ergeben können und durch welche Anpassungen sie gelöst werde können. Dazu wird auch ein Desing-Vorschlag für CAPPy erarbeitet, der seine funktionalen Eigenschaften erläutert. Darauf aufbauend wird gezeigt, wie durch eine Anbindung von ClioPatria in Python auch Rdf-Daten benutzerfreundlich verarbeitet werden können.

4.1 CAPJa – Ein Integrationsframework für Java und Prolog

Ostermayer [Ost17, S. 73-74] beschreibt CAPJa als ein Integrationsframework, das einen objektorientierten, einheitlichen Ansatz für die Integration von Prolog und Java besitzt und ohne Änderungen an der virtuellen Maschine von Java,

sowie den einzelnen PROLOG-Interpreter auskommt. Es stellt halb-automatische Mechanismen für die Integration von PROLOG-Prädikaten nach JAVA bereit und nutzt anonyme Funktionen in JAVA für Abfragen an PROLOG. Die Kommunikation zwischen PROLOG und JAVA basiert auf die vollautomatische Abbildung zwischen JAVA-Objekten und PROLOG-Termen. Durch ein erweiterbares System aus Gateways wird die Verbindung von JAVA zu verschieden PROLOG-Systemen hergestellt.

In diesem Abschnitt werden die Kernkomponenten von CAPJA eingeführt, die Konzepte von CAPJA analysiert und darauf aufbauend wird gezeigt, warum CAPJA eine ideale Vorlage für CAPPY ist.

4.1.1 Die Kernkomponenten in CAPJa

Die drei Kernkomponenten von CAPJA sind JPMAPPING, JPLAMBDA und JPGATEWAY. Sie werden im Folgenden detaillierter dargestellt.

JPMapping, die Abbildungskomponente. Nach Ostermayer [Ost17, S.81-82] ermöglicht die JPMAPPING-Komponente von CAPJA einen automatisierbaren und anpassbaren Abbildungsmechanismus zwischen JAVA-Objekten und PROLOG-Termen. So stellt es für JAVA mittels einer Standard-Abbildung einen Mechanismus bereit, der auf fast jeder JAVA-Klasse anwendbar ist und keinen zusätzlichen Programmieraufwand bedarf. Die Standard-Abbildung kann auch anhand von Annotationen im JAVA-Quellcode modifiziert werden. Dadurch kann der Benutzer die Repräsentation einer JAVA-Instanz als PROLOG-Term vollständig anpassen. Die Annotations-Schicht wird dabei lediglich für die Spezifikation der Abbildung genutzt und beeinflusst den Abbildungsprozess zur Laufzeit nicht. Dafür besitzt JPMAPPING einen Quellcode-Generator, der für jede JAVA-Klasse eine korrespondierende Abbildungsklasse erzeugt. Eine solche Abbildungsklasse übersetzt die Instanzen der JAVA-Klasse zu einem entsprechendem PROLOG-Prädikat und umgekehrt.

Des Weiteren [Ost17, S.81-82] stellt JPMAPPING durch die PSN (PREDICATE-SIGNATURE-NOTATION) einen Mechanismus für die halb-automatische Integration von PROLOG-Prädikaten nach JAVA bereit. Hierbei werden die einzelnen Prädikate in PROLOG mittels Signaturen beschrieben. Eine solche Beschreibung umfasst dabei die strukturelle Komposition eines Prädikats (So besteht ein PROLOG-Term aus einem Funktor, der Stelligkeit und den Argumenten). Der Quellcode-Generator von JPMAPPING benutzt eine PSN-Annotation, um einerseits eine Abbildungsklasse in JAVA zu generieren und um andererseits eine zusätzliche JAVA-Klasse zu generieren, die das PROLOG-Prädikat repräsentiert.

JPLambda, die Abfragekomponente. Nach Ostermayer [Ost17, S. 105-106] stellt JPLambda in Java einen eleganten und mächtigen Abfragemechanismus nach Prolog bereit. Hierfür nutzt JPLambda, aufbauend auf JPMapping, eine interne domänenspezifische Sprache in Java. Mit ihr können klare, präzise und objektorientierte Abfragen nach Prolog gestellt werden. Diese DSL heißt Java-Prolog Query Language (Jpql).

Eine zweite interne DSL ist die Java-Prolog Mapping Language (Jpml) Sie erlaubt dem Benutzer die Objekt-zu-Term-Abbildung in Java explizit zu modifizieren, in dem sie eine *Factory*-Klasse für die Abbildungen bereitstellt. Sie ist damit eine Alternative zu den JPMapping-Annotationen. Sie wird benötigt, wenn der Quellcode nicht verfügbar ist oder man auf ihn nicht zugreifen kann. Die Abbildungs-Spezifikationen werden ebenfalls mittels anonymer Funktionen in Jpml ausgedrückt.

Die beiden DSLs werden nur zur Spezifikation genutzt und besitzen kein Laufzeitverhalten. Der Quelle-zu-Quelle-Übersetzer JPCompiler analysiert die Spezifikationen in den ursprünglichen Java-Quelldateien und erzeugt neue Quelldateien. Im Anschluss modifiziert er die ursprünglichen Quelldateien durch eine Referenz auf die neu generierten Quelldateien, die die Abfragen an Prolog oder den benutzerdefinierten Abbildungen effizient implementiert.

JPGateway, die Verbindungskomponente. Nach Ostermayer [Ost17, S. 77-78] ist JPGateway ein System aus Gateways, welches die Kommunikation von Java mit verschieden Prolog-Systemen ermöglicht. Ein solches Gateway implementiert eine spezifische Kommunikation mit Prolog. Hierbei ist ein bestimmtes Standard-Gateway enthalten, das Portable Prolog Gateway (Ppg) heißt. Es nutzt keine Implementierungs-spezifischen Syntax und nutzt Standard-Streams für die Ein- und Ausgabe. Auf diesem Weg ist das Ppg bereits nativ mit vielen Prolog-Systemen kompatibel. Benutzerdefinierte Gateways können auch einfach hinzugefügt werden. Der Code für die einzelnen Gateways ist sauber von dem restlichen Java-Code getrennt. Daher kann jedes Prolog-System einfach ausgetauscht werden ohne das dafür größere Änderungen in Java nötig sind.

4.1.2 Analyse der Konzepte von CAPJa

In diesem Abschnitt werden die Konzepte von CAPJa abstrakt beschrieben, die die Integration von Prolog und Java in CAPJa ermöglichen.

Übertragung der Konzepte. In Java werden Daten in Objekten gespeichert und in Prolog in Termen, weswegen CAPJa Java-Objekte und Prolog-Terme aufeinander abbildet. Die Abbildung kann dabei vom Nutzer modifiziert werden.

Analyse mit Quellcode-Generierung. Damit die Abbildung durchgeführt werden kann, wird neuer Quellcode erzeugt. Die Funktionalität des neuen Codes ergibt sich durch die Analyse des bereits bestehenden Codes. Diese Analyse wird dabei automatisiert durchgeführt.

Abfrageoperationen. CAPJa bietet eine eigene Abfragesprache, die die folgenden Eigenschaften erfüllt:

Eingebettet Sie stellt eine interne DSL in Java dar

Deklarativ Sie beschreibt lediglich, *was* die Abfrage berechnen soll und nicht, *wie* die einzelnen Berechnungsschritte lauten

Kompakt Die Länge der Abfrage ist kurz

Intuitiv Sie kann ohne langwierige Einarbeitung von einem Java-Programmierer genutzt werden

Die Abfrageoperationen erfolgen in Java durch anonyme Funktionen mit bedingten und relationalen Operatoren. Durch Verwendung einer internen DSL können sie kompakt, lesbar und auf natürliche Art und Weise ausgedrückt werden.

Integration eines Prolog-Programms. Ein bestehendes Prolog-Programm kann in Java integriert werden, in dem von Hand das Prolog-Programm geändert wird. Hierbei müssen für jedes Prädikat, das nach Java abgebildet werden soll, Beschreibungen hinzugefügt werden, die seine Struktur beschreiben.

4.1.3 CAPJa als Vorlage für CAPPy

CAPJa ist folglich ein Software-Tool, mit dem es effizient möglich ist Prolog in Java einzubinden, wobei der Anwender keine spezifischen Prolog-Vorkenntnisse besitzen muss. Er muss lediglich nativen Java-Code schreiben. Dabei übernimmt CAPJa die komplette Interaktion mit Prolog. Weiterhin ist es auch unabhängig von einzelnen Prolog-Interpretern. CAPJa ist dadurch eine ideale Vorlage für ein neues Integrationsframework für Prolog und Python. Hierbei kann das Fra-

mework selbstverständlich nicht eins-zu-eins übertragen werden, sondern es müssen einige Anpassungen vorgenommen werden. Durch die Analyse der Konzepte ergibt sich aber, dass zumindest die Prinzipien übernommen werden können, die so eine solide Basis für ein neues Werkzeug darstellen.

4.2 Machbarkeitsstudie – Übertragung der Konzepte von CAPJa

In diesem Abschnitt wird dargelegt, inwiefern eine Übertragung der Synthese der Konzepte von CAPJa auf CAPPy möglich ist und welche Problemstellungen dabei auftreten können. Hierfür werden Lösungsvorschläge erarbeitet.

Dazu wird das funktionelle Design von CAPPy dargestellt. Im Einzelnen wird hierfür gezeigt, wie durch korrespondieren Strukturen in Prolog und Python eine einfachere Sichtweise auf Prolog-Prädikate ermöglicht wird, wie sie leichter durch Annotationen etabliert werden können und wie sie durch die objektorientierte Abfragesprache Ppql verarbeitet werden können, ohne Vorkenntnisse in Prolog besitzen zu müssen. Im Anschluss hieran wird ausgeführt, wie mittels CAPPy auch Rdf-Daten benutzerfreundlich verarbeitet werden können und welchen Nutzen CAPPy hierbei mit sich bringt. Auch wird untersucht, ob es für die in CAPJa genutzten Werkzeuge und Spracheigenschaften Äquivalente in Python existieren. Zusätzlich zeigt ein einfacher Prototyp die praktische Durchführbarkeit auf.

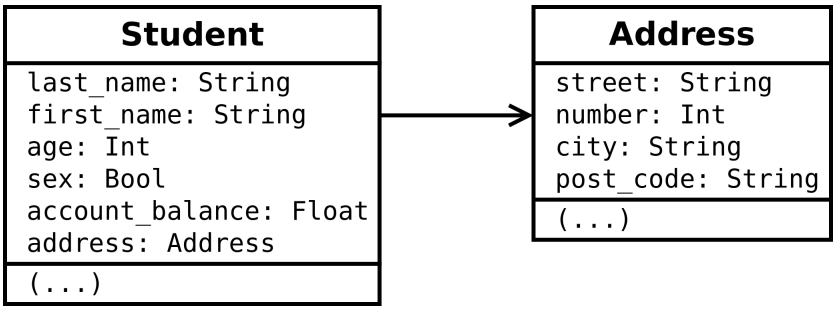
4.2.1 Funktionales Design von CAPPy

In diesem Abschnitt wird das funktionelle Design von CAPPy vorgestellt. Es wird gezeigt, wie anhand von korrespondierenden Strukturen in Prolog und Python eine intuitiv zugängliche Sichtweise auf Prolog-Prädikate erreicht werden kann. Weiterhin wird die neue Abfragesprache Ppql eingeführt. Mit ihr ist es möglich, Abfragen an eine Prolog-Datenbank einfach und kompakt zu stellen, ohne sich mit Prolog auskennen zu müssen. Weiterhin wird die Pmn vorgestellt. Mit ihr ist es möglich, Prädikate in Prolog so zu annotieren, dass sie auf aussagekräftigere Python-Strukturen abgebildet werden können. Insgesamt wird skizziert, wie der der Programmablauf in CAPPy vom Erstellen einer Abfrage bis hin zum Generieren der Ergebnisse funktioniert.

Listing 4.1: Universitätsdatenbank in Prolog

```
% student(Last_Name, First_Name, Age, Sex,
% Account_Balance, Address).
% address(Street, Number, City, Post_Code).
student('Mustermann', 'Max', 20, True, 123.45,
 ↪ address('Musterstrasse', 1, 'Musterhausen', 10000))
```

Abbildung 4.1: Klassen der Universitätsdatenbank in Python



Korrespondierende Strukturen in Prolog und Python. Es sei als Beispiel eine Universitätsdatenbank in Prolog gegeben. Ihre Struktur und die enthaltenen Fakten werden durch den Code in Listing 4.1 repräsentiert.

Es sei weiterhin ein Python-Programm gegeben, das eine dazu korrespondierende Klassenstruktur besitzt. Das Prädikat `student` wird durch die Klasse `Student` und das Prädikat `addresse` durch die Klasse `Addresse` repräsentiert. Dieser strukturelle Aufbau wird in dem Klassendiagramm in Abbildung 4.1 graphisch veranschaulicht. Durch die objektorientierte Repräsentation der Prolog-Datenbank erhält der Python-Entwickler eine intuitiv verständliche Sichtweise auf die Datenbankstruktur.

Auch die Fakten der Datenbank können durch Python-Strukturen repräsentiert werden, in dem für jedes Faktum ein korrespondierendes Objekt erstellt wird. Der Datenbankinhalt aus dem Listing 4.1 kann wie in Listing 4.2 in Python wiedergegeben werden. Der Sinn einer Datenbank liegt aber nicht darin, ihren Inhalt in ein Programm einfach komplett zu überführen und dort zu verarbeiten, sondern die persistente Speicherung und Verarbeitung soll durch das Datenbanksystem selbst realisiert werden. Mittels einer deklarativen Sprache werden Abfragen von einem Programm an die Datenbank gestellt, welche die Auswertung übernimmt und das Ergebnis an das Programm zurück gibt.

Listing 4.2: Universitätsdatenbank in Python

```
university_database =
 ↪ [Student('Mustermann', 'Max', 20, True, 123.45,
 ↪ Address('Musterstrasse', 1, 'Musterhausen',
 ↪ 10000))]
```

Listing 4.3: Definition einer Ppql-Abfrage

```
def queryName [types]: constraints
```

Ziel von CAPPy ist es anhand dieser korrespondierenden Struktur eine Möglichkeit zu schaffen, die es einem Python-Entwickler ermöglicht auf möglichst einfache und intuitive Weise eine Prolog-Datenbank abzufragen bzw. zu verändern.

Die Abfragesprache Ppql. Damit der Nutzer keine völlig neue Syntax erlernen muss, implementiert CAPPy eine interne DSL, die Abfragen an Prolog in Python ermöglicht. Sie heißt Prolog-Python Query Langauge, kurz Ppql, und ist an die Jpql [Ost17, S. 109-113] angelehnt, wobei sie syntaktisch auf Python angepasst wurde.

Eine Ppql-Abfrage ist syntaktisch korrekter Python-Code. Daher kann es zu keinen Interpretations- bzw. Kompilierungsfehlern kommen. Allerdings wird er nicht direkt ausgeführt, sondern dient vielmehr zur Spezifikation der Abfrage. Ppql stützt sich dabei auf die korrespondierende Klassen- und Prädikatenstruktur. Intuitiv kann eine Ppql-Abfrage so verstanden werden, dass der Programmierer festlegt, welche Eigenschaften die Objekte der korrespondierenden Klassen erfüllen sollen. Daraus formuliert er eine Python-Funktion, dessen Parameter die gesuchten Objekte sind und in dessen Funktionsrumpf die Bedingungen formuliert werden, die diese Objekte erfüllen sollen. CAPPy werte die Abfrage aus und liefert als Ergebnis die Objekte, die die spezifizierten Eigenschaften erfüllen. Da die Abfragen über Objekte formuliert werden, ist Ppql eine objektorientierte Abfragesprache. Die Syntax einer Abfrage in Ppql wird in Listing 4.3 defniert. `queryName` ist dabei der Name der Abfrage. Für die syntaktischen Eigenschaften des Namens gelten die selben Bedingungen wie für gewöhnliche Funktions-Bezeichner in Python. `[types]` repräsentiert die Abfragetypen, welche durch Kommata getrennt werden. Hierbei muss für jeden Abfragetyp ein korrekter Typhinweis angegeben werden. Ein Abfragetyp hat damit die die Syntax `objectName : Class`. Dabei

Listing 4.4: Einfache Abfrage in Ppql

```
1 def studentQuery1 (s : Student):
2   s.last_name == "Mustermann" and s.age >= 18
```

ist `objectName` der Name des Abfragetyps und `Class` der Bezeichner der Klasse, dessen Instanz der Abfragetyp ist. Für die syntaktischen Eigenschaften von `typeName` gelten die selben Bedingungen, wie für gewöhnliche Attributs-Bezeichner in Python. Werden mehrere Abfragetypen definiert, so ist die Reihenfolge an sich trivial. Lediglich die Reihenfolge der Rückgabewerte wird dadurch festgelegt.

Die Bedingungen werden mittels der Vergleichsoperatoren <, >, >=, <= , == und ! = über die Attribute der Abfragetypen oder hartkodierter Werte der Python-Typen `int`, `long`, `float` oder `complex` gebildet. Bei den Python-Typen `str` und `bool` sind nur Tests auf Gleichheit und Ungleichheit möglich, daher == und ! =. Bei den Container-Typen von Python `list`, `tupel`, `set`, und `frozenset` kann mittels `in` bzw. `not in` abfragt werden ob ein bestimmter Wert enthalten ist bzw. nicht enthalten ist. Wenn `dc` vom Python-Typ `dict` ist und `i` ein Schlüssel in `dc` ist, dann kann mit `dc[i]` der Eintrag aus `dc` für den Schlüssel `i` abgefragt werden. Alle Bedingungen können Python-typisch mittels `and` und `or` verknüpft werden. Durch ein direkt vorangestelltes `not` kann eine Negation durchgeführt werden. Durch runde Klammern kann eine Auswertungsreihenfolge bestimmt werden.

Ppql besitzt noch den reservierten Bezeichner `Omit`, der an beliebiger Stelle in eine Abfrage miteingebaut werden kann. Durch `omit(qt.a)` kann angegeben werden, dass das Attribut `a` von dem Abfragetyp mit dem Bezeichner `qt` nicht von Interesse ist. Dadurch kann die Auswertung beschleunigt werden. Es wird in diesem Fall der Wert `none` für das bezeichnete Attribut im Ergebnisobjekt eingetragen. `Omit` wird nicht durch Bedingungen in der Abfrage beeinflusst.

Beispiele in Ppql. Angenommen ein Benutzer möchte eine Abfrage stellen, die alle Studenten bestimmt, die mit Nachnamen *Mustermann* heißen und mindestens 18 Jahre alt sind. Mithilfe des korrespondierenden Klassenmodells, kann sie wie folgt formalisiert werden: Bestimme alle `Student`-Objekte, deren Attributausprägung des Attributs `last_name` gleich "Mustermann" ist und deren Attributausprägung des Attributs `age` größer-gleich 18 ist.

Da nach einer Instanz der Klasse `Student` gefragt ist, ist der Abfragetyp von der Klasse `Student`. Die zuvor eingeführte Abfrage lässt sich also wie in Listing 4.4 in Ppql ausdrücken. Diese Abfrage kann der Nutzer CAPPy übergeben und

Listing 4.5: Ausführen der Abfrage aus Listing 4.4

```
1 cappy.findAllSolutions(studentQuery1)
```

Listing 4.6: Ergebnis der Abfrage aus Listing 4.4

```
1 result = [(Student('Mustermann', 'Max', 20, True, 123.45,
2  ↪ Address('Musterstrasse', 1, 'Musterhausen',
3  ↪ 10000)))]
```

anweisen, dass alle passenden Lösungen gefunden werden sollen, siehe Listing 4.5. Das Ergebnis soll eine Liste aus Tupeln sein. Jedes dieser Tupel enthält ein Objekt der Klasse `Student`, auf dem die Abfragespezifikation zutrifft. Im konkreten Fall ist das Ergebnis eine Liste mit einem einzigen Tupel, der den einzigen Studenten enthält, der in der Universität-Datenbank enthalten ist, siehe Listing 4.6

Alle Studenten, deren Geschlecht männlich ist oder in Würzburg wohnen, lassen sich durch eine Abfrage, wie in Listing 4.7 bestimmen. Wenn der Nachname in dieser Abfrage nicht von Bedeutung ist, kann die Abfrage wie in Listing 4.8 durch ein `Omit` erweitert werden.

Auswertung einer Ppql-Abfrage. Eine Abfrage in PPQL wird nie als solcher direkt in PYTHON ausgeführt. Stattdessen wandelt die Übersetzungs-Komponente von CAPPy sie in eine äquivalente PROLOG-Abfrage um. Dazu verarbeitet sie die Abfrage mittels eines Parser-Generators.

Die Spezifikation einer Abfrage in PPQL kann für verschiedene Arten der Datenbankinteraktion genutzt werden. Konkret stehen dem Nutzer folgende Möglichkeiten in CAPPy offen:

- Er kann sich eine einzelne Instanz, auf die die Abfrage passt, ausgeben lassen und bei Bedarf weitere Instanzen anfordern (Vgl. Backtracking in PROLOG)
- Er kann sich alle Instanzen, auf die die Abfrage passt, ausgeben lassen

Listing 4.7: Weitere Abfrage in PPQL

```
1 def studentQuery2 (s : Student):
2   s.sex == True or s.address.city == "Wuerzburg"
```

Listing 4.8: PPQL-Abfrage mit `omit`

```
def studentQuery3 (s : Student):
  s.sex == True or s.address.city == "Wuerzburg"
   ↪ and omit(s.last_name)
```

Listing 4.9: PROLOG-Übersetzung der Abfrage aus Listing 4.4

```
student(Student_0, Student_1, Student_2, Student_3,
 ↪ Student_4, Student_5), Student_0 = 'Mustermann',
 ↪ Student_2 >= 18.
```

- Er kann alle Instanzen in der Datenbank, die auf die Abfrage passen, löschen lassen

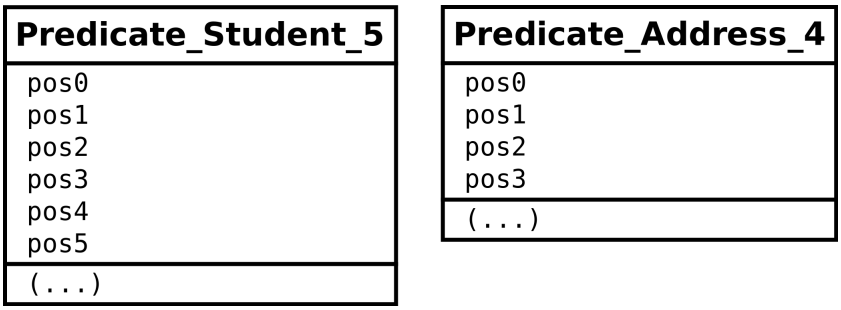
Die Abfrage `studentQuery1` aus Listing 4.4 könnte zum Beispiel wie in Listing 4.9 in eine PROLOG-Abfrage umgewandelt werden.

Übermittlung von Abfragen durch ein Gateway. Eine nach PROLOG übersetzte Abfrage wird von CAPPy an ein Gateway weitergeleitet. Dieses etabliert eine Verbindung zu einem PROLOG-Interpreter. In CAPPy ist das Gateway-Interface mit beliebigen PROLOG-Systemen kompatibel. Dazu ist es nötig, dass es nur solche Schnittstellen-Konstrukte verwendet, die alle PROLOG-Interpreter unterstützen. Welche Konstrukte dies sind, wurde in einer *ISO*-Norm standardisiert (siehe [ISOa]), weshalb diese Form auch ISO-PROLOG genannt wird. Das Gateway-Interface besitzt hierfür, wie in CAPJA [Ost17, S. 125ff], eine Klasse, die `PPG` heißt, die als Schnittstelle für das restliche Programm zum Gateway fungiert.

Für jeden von CAPPy unterstützten PROLOG-Interpreter existiert eine Unterklasse, die mit einem bestimmten PROLOG-Interpreter interagieren kann. CAPPy bietet zunächst nur eine Unterstützung von SWI-PROLOG an. Die Unterstützung dieses Interpreters ist wichtig, weil er einerseits weit verbreitet ist und andererseits auch CLIOPATRIA unterstützt. Eine Unterklasse von `PPG` kommuniziert die Abfrage und bereitet das Abfrageergebnis aus PROLOG vor, indem sie die Antwort in einfache PYTHON-Konstrukte überführt.

Überführen des Abfrageergebnisses in Python-Strukturen. Nachdem das Gateway die Abfrage an einen PROLOG-Interpreter übergeben hat und die Rückgabe vor-verarbeitet hat, wird das Ergebnis für das PYTHON-Programm aufgearbeitet.

Abbildung 4.2: Direkt generierte PYTHON-Klassen



Sollten Instanzen gelöscht werden, so wird mittels eines Wahrheitswertes angezeigt, ob die Durchführung erfolgreich war. Wurden nach konkreten Instanzen gefragt, so wird für jeden Abfragetyp ein Objekt erzeugt und seine Attribute auf die Ergebniswerte gesetzt. Dies wird von der Übersetzungskomponente durchgeführt, die dazu die Abbildungskomponente anfragen muss, um die Ergebniswerte den richtigen Attributen zuordnen zu können. Die generierten Objekte werden dabei in Tupeln zusammengefasst. Die Reihenfolge im Tupel entspricht dabei der Reihenfolge der Abfragetypen in der Abfrage. Dies ist eine PYTHON-typische Struktur, die ein PYTHON-Entwickler intuitiv verstehen und verarbeiten kann.

Etablierung der korrespondierenden Strukturen in Python mithilfe der Pmn. Die eingangs erwähnte korrespondierende Struktur in PYTHON von Hand zu erstellen wäre sehr aufwendig. CAPPy bietet daher die Möglichkeit PROLOG-Strukturen einzulesen. Es analysiert dabei die enthaltenen Prädikate und bildet sie auf eine Klassenstruktur ab. Betrachte als Beispiel erneut die Universitätsdatenbank in Listing 4.1. Diese Struktur kann einfach in Klassen umgewandelt werden, wie in Abbildung 4.2 dargestellt. Ein Klassenname besteht dabei unter anderem aus dem Funktor des Prädikats und der Stelligkeit des Prädikats. Die Angabe der Stelligkeit ist wichtig, da Prädikate mit selben Funktor aber unterschiedlicher Stelligkeit in PROLOG als unterschiedliche Strukturen gelten.

In dieser Abbildung sind die Attribute nicht benannt und es existieren auch keine Typhinweise. Diese Struktur ist somit schwer zugänglich oder zumindest benutzerfreundlich. Damit aussagekräftigere Klassen erstellt werden können, müssen die PROLOG-Prädikate annotiert werden. CAPJA setzt hierfür die PSN ein, was für PREDICATE SIGNATURE NOTATION steht. Mit ihr kann eine Signatur eines PROLOG-Prädikats kompakt und natürlich ausgedrückt werden. Als eine Abwandlung hiervon unterstützt CAPPy die PMN, was für PROLOG MAPPING NOTATION

Listing 4.10: Struktur einer PMN-Notation

```
1 predicate(name, [bezeichner], [typen]).
```

steht. Ihr Ziel besteht darin eine einfachere Syntax anzubieten und auch Bezeichner in natürlicher Sprache zu ermöglichen, so dass die Prädikate leichter in beliebige objektorientierte Programmiersprachen abgebildet werden können.

Die grundsätzliche Struktur einer Notation in PMN findet sich in Listing 4.10. Dabei ist `name` der Name des PROLOG-Prädikats, das annotiert werden soll. `[bezeichner]` ist eine Liste mit der Länge, die der Stelligkeit des zu annotierenden PROLOG-Prädikats entspricht. In der Reihenfolge, in der die Komponenten in dem Prädikat auftreten, gibt er für jede Komponente an, wie er in natürlicher Sprache beschrieben werden kann. Die erlaubten Zeichen werden durch den reguläre Ausdruck `'([a-z][0-9]_)+'` beschrieben. Wird das Prädikat auf eine Klasse abgebildet, so wird die Beschreibung als Attributsbezeichner benutzt, wobei die umschließenden Hochkommata ignoriert werden. `[typen]` ist eine Liste mit der Länge, die der Stelligkeit des zu annotierenden PROLOG-Prädikats entspricht. In der Reihenfolge, in der die Komponenten in dem Prädikat auftreten, gibt es für jede Komponente an, welchem Typ ein Wert in dieser Komponente entspricht. In der PMN wird der Nutzer folglich darauf eingeschränkt, nur einen bestimmten Typ für eine bestimmte Komponente zu benutzten. Für jeden Typ existiert in der PMN eine festgelegte Notation. Sie wurde so gewählt, dass sie sich an der nativen PROLOG-Schreibweise und der Schreibweise in bekannten objektorientierten Programmiersprachen orientiert. Die einzelnen Notationen werden in der Tabelle 4.1 definiert.

Durch eine Annotation eines PROLOG-Prädikats in PMN können bessere Abbildungsmechanismen erschaffen werden. Sei die PMN des `student`-Prädikats in der Universitätsdatenbank wie in Listing 4.11 gegeben. Die generierte Klassenstruktur daraus kann in Abbildung 4.3 eingesehen werden. Sie ist offensichtlich zugänglicher als die Klassenstruktur, die ohne PMN-Annotation generiert werden kann (Vgl. Abbildung 4.2).

Etablierung der korrespondierenden Strukturen in Prolog. CAPPy bietet weiterhin die Möglichkeit PYTHON-Klassen auf ein PYTHON-Prädikat abzubilden. Ein PYTHON-Programmierer kann dadurch sogar eine PROLOG-Datenbank erstellen, indem er CAPPy eine Menge von Objekten übergibt, aus denen es korrespondierende Terme generiert und sie durch einen PROLOG-Interpreter zusichern lässt. Die Etablierung der korrespondierenden Strukturen in PROLOG erfolgt da-

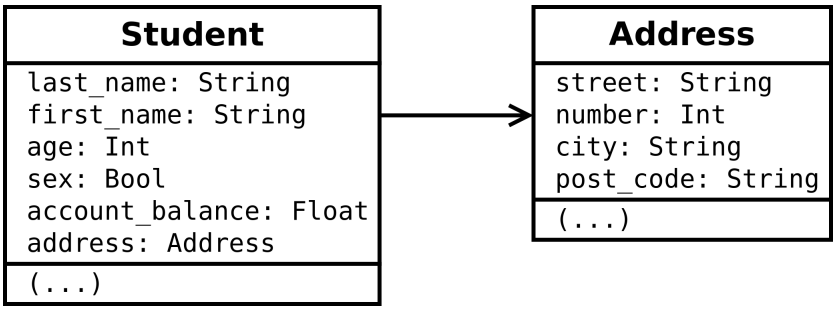
Listing 4.11: PMN-Notation des Prädikats `student`

```
predicate(student, ['last_name', 'first_name',
 ↪ 'age', 'sex', 'account_balance', 'address'],
 ↪ [str, str, int, bool, float,predicate(address)]).
predicate(address,
 ↪ ['street', 'number', 'city', 'post_code'],
 ↪ [str, int, str, str]).
```

Tabelle 4.1: Typen-Bezeichner in der PMN

Bezeichner	**Typ**
int	Ganze Zahl
float	Fließkommazahl
bool	Boolescher Ausdruck
str	Zeichenkette
list(type)	Liste, die nur Elemente des Typs `type` enthält
predicate(p)	PROLOG-Prädikat `p`

Abbildung 4.3: Aus der PMN-Notation generierte PYTHON-Klassen



Listing 4.12: `Student`-Objekt `s1`

```
s1 = Student('Mustermann', 'Max', 20, True, 123.45,
 ↪ Address('Musterstrasse', 1, 'Musterhausen', 10000))
```

Listing 4.13: Prolog-Repräsentation des Objekts `s1`

```
student('Mustermann', 'Max', 20, True, 123.45,
 ↪ address('Musterstrasse', 1, 'Musterhausen', 10000))
```

bei analog zur Etablierung der korrespondierenden Strukturen in Python. Die Abbildungskomponente muss lediglich festlegen, auf welcher Stelle in dem Prolog-Prädikat welches Attribut abgebildet werden soll.

Wird CAPPy dazu genutzt eine Prolog-Datenbank aufzubauen oder zu erweitern, so stellt die Aktualisierung der Datenbank ein Problem dar. Ist die Attributsausprägung eines Objektes `a` ein anderes Objekt `b`, so hat die Aktualisierung des Objekts `b` auch eine Auswirkung auf die Repräsentation des Objekts `a`. Sei beispielsweise die Klassenstruktur aus Abbildung 4.1 gegeben. Ein `Student`-Objekt `s1` habe im Attribut `address` die Ausprägung eines `Address`-Objektes, wie im Listing 4.12. Dies würde in die Prolog-Struktur aus Listing 4.13 überführt werden. Wenn nun das `Address`-Attribut aktualisiert werden soll indem, zum Beispiel, die Hausnummer auf `20` gesetzt wird, dann könnte der Benutzer nicht einfach CAPPy anweisen, den `address`-Term zu ändern, denn die Information müsste in dem `student`-Term geändert werden. Eine Aktualisierung der Datenbank durch das `Address`-Objekt müsste folglich zurückgewiesen werden. Sei zusätzlich angenommen, dass `Address`-Objekt wäre nicht nur eine Attributsausprägung von `s1` sondern noch Attributsausprägung eines anderen `Student`-Objektes `s2`, das ebenfalls auf einen Prolog-Term abgebildet wurde. Dann würde es in diesem Fall nicht ausreichen, wenn nur das Objekt `s1` zur Aktualisierung übergeben werden würde, sondern es müsste auch das Objekt `s2` übergeben werden. Dies würde für einen Python-Entwickler unerwartet sein und birgt die Gefahr von Inkonsistenzen zwischen den korrespondierenden Strukturen. CAPPy bietet daher keine direkte Möglichkeit zum Aktualisieren der Datenbank an. Es können lediglich neue Objekte inklusive aller referenzierter Strukturen in die Datenbank eingefügt oder gelöscht werden.

Bewertung des Designs. Durch das funktionelle Design von CAPPy entstehen Nutzer CAPPy eine Reihe von Vorteilen, die er sonst nicht genießen könnte:

- Ein Python-Programmierer muss kein Prolog erlernen, um Prolog nutzen zu können, denn CAPPy ist ein Werkzeug, dass ihn bei seinen bekannten Konzepten aus Python abholt und dadurch den Aufwand, eine völlig neue Abfragesprache zu erlernen, erspart.
- Ein weiterer Vorteil ist, dass man eine neue Programmiersprache ansprechen kann, ohne dass sie im Quellcode auftaucht und die Verständlichkeit beeinträchtigt.
- Durch die Automatisierungen in CAPPy müssen keine Konvertierungsmechanismen selber erstellt werden. Für jede Klasse bzw. Prädikat einen eigenen Abbildungsmechanismus zu erstellen ist unproduktiv. Daher automatisiert CAPPy dies und nimmt dem Programmierer die Arbeit ab oder bietet halb-automatische Lösungen an
- Selbst Features in Prolog, die gewöhnlich Prolog-Kenntnisse voraussetzen, wie das `findall`-Metaprädikat oder die Variablenunifikation, können ohne Vorkenntnisse genutzt werden.
- Auch kann der Prolog-Interpreter prinzipiell ausgetauscht werden, ohne dass dies Auswirkungen auf den Programmcode hat.

4.2.2 Verarbeitung von Rdf-Daten

Durch die einfache Integration von Prolog in Python können nicht nur Prolog-Datenbanken eingebunden werden, sondern auch Prolog-Anwendungen. In diesem Abschnitt wird gezeigt, wie sich mithilfe CAPPy auch Rdf-Daten einfach verarbeiten lassen, indem ClioPatria in Python integriert wird.

Einbindung von ClioPatria. ClioPatria ist eine Rdf-Bibliothek für Prolog. Sie kann unter anderem effizient Rdf-Daten persistent speichern. Indem durch CAPPy ClioPatria angesprochen wird, ergibt sich dadurch eine Möglichkeit auch Rdf-Daten in Python zu verarbeiten. Die Vorteile von CAPPy bestehen auch in dieser Hinsicht uneingeschränkt weiter. Insbesondere können Rdf-Daten kompakt und intuitiv verarbeitet werden.

Die Rdf-Komponente in CAPPy. Ein zusätzliches Modul in CAPPy übernimmt die komplette Interaktion mit ClioPatria. Der Nutzer muss lediglich bei CAPPy eine Operation aufrufen, wie der Code in Listing 4.14 zeigt. Dadurch wird

Listing 4.14: Starten und Beenden eines Rdf-Servers in CAPPy

```
# start rdf server
cappy.start_RDF_Server()
```

Listing 4.15: Laden einer Rdf-Datein in CAPPy

```
cappy.laod_rdf_file("path/to/rdffile")
```

ein Rdf-Server gestartet. Anschließend können Rdf-Dateien geladen werden, siehe Listing 4.15. Eine Datei muss dabei in den Formaten Rdf/Xml oder Turtle vorliegen, was typische Formate zur Speicherung von Rdf-Daten sind. CAPPy leitet den Ladebefehl an ClioPatria weiter

Um die Verarbeitung der Präfixe zu erleichtern, können vorab Präfixe dauerhaft definiert werden, siehe Listing 4.16. Dabei ist `uri` ein String, der eine Uri repräsentiert und `prefix` ein String, der als Alias für den Namensraum dienen soll. CAPPy leitet den gesetzten Alias an ClioPatria weiter, welches die nötige Auflösung vornimmt

In ClioPatria wird ein Rdf-Tripel durch das `rdf`-Prädikat repräsentiert. In Listing 4.17 ist seine Signatur und ein Beispielfakt angegeben. Analog, wie auch bei anderen Prolog-Strukturen, ergibt sich dadurch die in Listing 4.4 dargestellte korrespondierende Python-Klasse. Konkrete Rdf-Tripel werden daher durch Objekte mit den Attributen `subjektiv`, `prädikat` und `objekt` repräsentiert.

Rdf-Abfragen in Ppql. Häufig werden Rdf-Abfragen mittels einer speziellen Sprache, wie Sparql, gestellt. Im Folgenden wird ein Beispiel einer Sparql-Abfrage betrachtet, siehe Listing 4.18. Die einzelnen Zeilen haben dabei die folgende Bedeutung. In Zeile 1 wird ein Präfix definiert. In diesem Fall wird der Uri `http://xmlns.com/foaf/0.1/` der Alias `foaf` zugewiesen, womit sich die folgende Abfrage kompakter formulieren lässt. In Zeile 2 werden die Bezeichner der gesuchten Typen gesetzt. In Zeile den Zeilen 3 bis 5 werden die Bedingungen definiert,

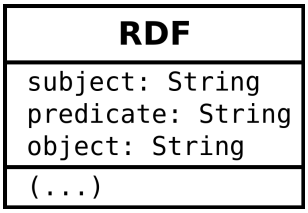
Listing 4.16: Registrierung eines Alias in CAPPy

```
cappy.register_namespace(prefix, uri)
```

Listing 4.17: RDF-Tripel in PROLOG

```
%rdf(Subject, Predicate, Object)
rdf('http://beispiel.de/lebensmittel/apfel',
 ↪ 'http://beispiel.de/praepositionen/ist_ein',
 ↪ 'http://beispiel.de/kategorien/Obst')
```

Abbildung 4.4: PYTHON-Repräsentation eines RDF-Tripels



die die Typen erfüllen müssen. Es ist evident, dass diese Form der Abfrage für einen PYTHON-Programmierer nicht intuitiv zugänglich ist.

Um einem PYTHON-Programmierer den Aufwand zu ersparen SPARQL zu erlernen, kann er CAPPy mit seiner PPQL verwenden. Wird eine Abfrage ausdrücklich als Abfrage über RDF-Tripel an CAPPy übergeben, so können dabei einige zusätzliche Features genutzt werden:

- Die Abfragetypen sind immer Objekte der Klasse `RDF`. Daher müssen lediglich die Bezeichner als Abfragetypen angegeben werden. Auf eine explizite Deklaration als Objekte der Klasse RDF mittels Typhinweise kann verzichtet werden.
- Die Präfixe werden als Operationen repräsentiert. Ist `abkz` ein bereits registriertes Präfix und `attr` ein Attribut eines `RDF`-Objektes, so bedeutet `abkz(attr)`, dass `attr` zum Namensraum `abkz` gehört. Dies würde in RDF-typischer Schreibweise als `abkz:attr` formuliert werden. Diese Syntax ist in

Listing 4.18: Abfrage der Emailadressen in SPARQL. Quelle: [ISOb]

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name ?mbox
WHERE
{ ?x foaf:name ?name .
?x foaf:mbox ?mbox }
```

Listing 4.19: Abfrage der Emailadressen in Ppql

```
1 def getMailboxes (name, mailBox):
2 name.subject == mailBox.subject and
3  ↪ name.predicate == foaf(name) and
4  ↪ mailBox.predicate == foaf(mailBox)
```

Python jedoch nicht möglich, da keine Doppelpunkte in den Bezeichnern erlaubt sind.

Die vorherige Sparql-Abfrage kann in Ppql wie in Listing 4.19 formuliert werden. Alleinig mit dem Wissen, was ein Rdf-Tripel ist, kann ein Python-Programmierer durch die Abfrage bereits erahnen, was die Abfrage zu bedeuten hat, was bei der Sparql Abfrage wahrscheinlich nicht ohne Weiteres der Fall wäre: Gesucht ist ein Paar von Rdf-Tripel, die die Namen und die Emailadressen einer Person repräsentieren. Dies demonstriert, wie intuitiv zugänglich die Ppql ist.

Im Gegensatz zu einer Sparql-Abfrage kann in einer Ppql-Abfrage keine einzelnen Werte abgefragt werden, sondern nur komplette Rdf-Tripel. Diese Form hat den Vorteil, dass sich die Abfragen so kompakter formulieren lassen.

Die Abfrageauswertung wird von den gleichen CAPPy-Komponenten geleistet, die auch für gewöhnliche Prolog-Abfragen zuständig sind. Lediglich die Übersetzungskomponente muss die zwei Änderungen an der Sprachspezifikation beachten.

Vorteile. Durch die Nutzung von CAPPy kann ein Python-Benutzer auf einfache, intuitive und kompakte Weise Abfragen an Rdf-Daten stellen. Insbesondere muss er keine völlig unterschiedliche Sprache, wie Sparql oder SeRQL erlernen. Rdf-Abfragen in Ppql sind leicht zu verstehen und ihre Erstellung kann schnell erlernt werden. Dies führt zur Vermeidung von Programmierfehlern und erleichtert die Wartung des Programmcodes erheblich.

4.2.3 Unterschiede zwischen Python und Java

In diesem Abschnitt werden die für die Implementierung von CAPPy relevanten Unterschiede zwischen Python und Java herausgearbeitet. Darauf aufbauend werden die Konsequenzen diskutiert, die daraus für die Umsetzung von CAPPy resultieren.

Objektorientierung. Java besitzt einen strengen objektorientierten Ansatz. Bereits für ein "Hello World"-Programm muss eine Klasse geschrieben werden. Python hingegen enthält ebenfalls objektorientierte Elemente, doch es lässt dem Programmierer auch den Freiraum hierauf zu verzichten. Um den Klassenmechanismus in Python verwenden zu können, bedarf es nur eines minimalen Aufwands, um die entsprechende Syntax und Semantik zu erlernen. Er besitzt typische Eigenschaften des objektorientierten Programmierens, wie Vererbung mit überschreiben und aufrufen von Methoden der Oberklasse (Vgl. [vPc, S. 67]). Objektorientierung ist folglich eine natürliche Eigenschaft von Python.

In Java ist keine Mehrfachvererbung möglich. Als Alternative für Mehrfachvererbung können in Java Interfaces verwendet werden. Eine Java Klasse kann eine beliebige Anzahl an Interfaces integrieren. Ein Interface definiert dabei lediglich die Signatur der Methoden. Über *Getter*- und *Setter*-Methoden (sogenannte Java Beans) können in einem Interface die Attribute einer Klasse indirekt deklariert werden (Vgl. [Ull, Kap. 5.13]). In Python hingegen existieren keine Interfaces. Es ermöglicht dafür die Mehrfachvererbung, womit der Bedarf an Interfaces obsolet erscheint (Vgl. [vPc, S. 76]).

Daraus folgt, dass Java eine rein objektorientierte Programmiersprache ist, während in Python die Objektorientierung nur eine Möglichkeit darstellt, die Sprache zu nutzen. Da in Java jeder Typ gleichzeitig ein Objekt ist, genügt es in CAPJa sich auf Objekte zu beschränken. Bei der Integration von Prolog und Python muss daher festgelegt werden, wie Python-Typen abgebildet werden sollen, die keine Objekte sind.

Zunächst lässt sich feststellen, dass sich das Problem nicht ergibt, wenn eine Prolog-Struktur auf eine Python-Struktur abgebildet wird, wenn die Implementierung von CAPPy so gestaltet ist, dass es die Prolog-Strukturen stets auf Objekte bzw. Klassen abgebildet, was bei relationale Einheiten immer möglich ist. Für den umgekehrten Fall bieten sich zwei Lösungsvorschläge an:

- Die Benutzung von CAPPy wird darauf eingeschränkt, dass lediglich Objekte auf Prolog-Strukturen abgebildet werden können. Das Ziel von CAPPy ist es, eine deduktive Datenbankkomponente für Python-Entwickler bereitzustellen. Einzelne Variablen lassen sich hierbei allerdings nur bedingt sinnvoll einsetzen. Denn alle Objekte einer Klasse können als Terme über das selbe Prädikat aufgefasst werden, was bei einzelnen Variablen nicht sinnvoll möglich ist. Einzelne, unstrukturierte Werte sind in einer Datenbank fehl am Platz. Weil die Objektorientierung ein natürliches Prinzip in Python ist und somit jeder Python-Programmierer damit umgehen kann, kann diese Einschränkung gerechtfertigt sein.

- Wenn einzelne Werte nach PROLOG abgebildet werden sollen, so können speziell hierfür reservierte Prädikate eingesetzt werden. Diese beschreiben zu welcher Datei der Wert gehört bzw. dass er global ist, wie sein Bezeichner lautet und welche Ausprägung er hat.

Für die Implementierung von CAPPy wurde der erste Lösungsvorschlag gewählt. Auf die Übersetzungskomponente haben die nicht objektorientierten Elemente in PYTHON keinen Einfluss, denn eine PPQL-Abfrage kann definitionsgemäß nur über Objekte und einfacher, hartkodierter Werte erfolgen.

Dynamische Erweiterbarkeit. In PYTHON kann zur Laufzeit das Programm erweitert werden. Dies ist in JAVA zwar auch möglich, es Bedarf hierfür aber fortgeschrittener Programmierkenntnisse. So kann, zum Beispiel, dem JAVA-*Compiler* zur Laufzeit Code zum Kompilieren übergeben werden. Die meisten JAVA-Programme verzichten jedoch auf solch dynamische Änderungen, die eher ein Randphänomen für spezielle Zwecke darstellen. In PYTHON hingegen gilt die dynamische Erweiterbarkeit als ein natürliches Prinzip und gehört zum Standardrepertoire eines PYTHON-Programmierers.

So können in PYTHON Klassen leicht zur Laufzeit erschaffen und modifiziert werden (Vgl. [vPc, S. 67]). Selbst aus der Klassendefinition ergibt sich keine eindeutige Attributmenge eines Objektes. Denn ein Objekt kann sich zur Laufzeit selbst dynamisch neue Attribute geben. Darüber hinaus kann sogar von beliebiger externer Stelle ein Objekt um beliebig viele neue Attribute erweitert werden. Dies muss bei dem Design von CAPPy berücksichtigt werden.

Für das Problem, wie mit Klassen umgegangen werden soll, die zur Übersetzungszeit nicht bekannt sind, bieten sich zwei Lösungsvorschläge an:

- Es können, wie bei CAPJA, Annotationen in Form einer internen DSL eingesetzt werden, mit der beschrieben werden kann, wie Klassen, die zur Übersetzungszeit noch nicht bekannt sind, abgebildet werden sollen
- Bei einer dynamischen Abbildungskomponente tritt dieses Problem nicht auf. Sie wird erst zur Laufzeit aktiv und bildet alle Objekte bzw. Terme dynamisch ab, auch wenn keine Abbildungsvorschrift bekannt ist.

Für das Problem, dass die Attributmenge einer PYTHON-Klasse nicht eindeutig bestimmt werden kann, bieten sich folgende Lösungsvorschläge an:

- Zunächst ergibt sich die Möglichkeit der Restriktion. Der Nutzer von CAPPy wird darauf eingeschränkt nur eine festgelegte Menge von Attributen in einer

Listing 4.20: PROLOG-Represäntation mit Attributbeschreibung

```
1 student(['Name','Alter','Fach'], 'Schneider',
2  ↪ 20, 'Informatik')
```

Klasse zu nutzen. Eine Möglichkeit hierfür wäre es, dass nur im Konstruktor gesetzte Attribute genutzt werden dürfen.

- Eine Erweiterung zu der ersten Lösungsmöglichkeit ist es, dass in den Fällen, in denen ein unerwartetes Attribut bei der Abbildung auftaucht, der Nutzer das Programm so um deklarative Anweisungen (z. B. durch eine interne DSL) erweitern muss, dass CAPPy mit diesem Sonderfall umgehen kann. Ansonsten würde das Objekt nicht abgebildet werden.

- Eine Alternative wäre es, jedem neu erstellten PROLOG-Prädikat eine Beschreibung mitzugeben, was die einzelne Komponenten bedeuten. Dies könnte z. B. geschehen, indem an der ersten Stelle des Prädikats immer eine Liste mit den Bezeichnern eingefügt wird. Ein Objekt von der Klasse `Student` mit den Attributausprägungen `Name = Schneider`, `Alter = 20` und `Studienfach = Informatik` würde dann auf ein PROLOG-Prädikat wie in Listing 4.20 abgebildet werden. Diese Darstellung hat den Vorteil, dass sie sehr flexibel mit den unterschiedlichsten Varianten eines Objekts umgehen kann. Ein bedeutender Nachteil ist aber, dass in dieser Variante ein PROLOG-Prädikat nur noch ein einfacher Informationsträger ist. In PROLOG wird eigentlich durch die Stelligkeit angegeben, was eine Komponente in einem Term zu bedeuten hat. Dieser Zusammenhang fehlt an dieser Stelle, so dass der erzeugte Code als atypischer PROLOG-Code angesehen werden muss. Insbesondere können bereits existierende PROLOG-Programme aus diesem Grund mit dieser Art der Repräsentation in den allermeisten Fällen nicht umgehen.

- Auch wenn PYTHON es zulässt, dass Attribute dynamisch hinzugefügt werden können, so kann man dies in Hinblick auf das Paradigma der objektorientierten Programmierung als nicht-erwünschten Programmierstil betrachten. Daher sollte dieser Fall in der Regel nicht vorkommen und CAPPy muss unerwartete Attribute nur als Ausnahmefall abdecken. Die Abbildungsvorschrift für eine Klasse wird dynamisch zur Laufzeit erzeugt, sobald eine Instanz dieser Klasse erstmals abgebildet werden soll. Die erste Instanz dient dabei als Prototyp für alle Instanzen dieser Klasse. Sie wird analysiert und aus dem Analyseergebnis wird eine Abbildungsvorschrift generiert. Es wird dann davon ausgegangen, dass es sich bei der Attributmenge des Objektes um die einzige mögliche Attributmenge aller Objekte dieser Klasse handelt.

Listing 4.21: Prolog-Represäntation ohne Attributbeschreibung

```
1 student('Schneider', 20, 'Informatik').
```

Listing 4.22: Prolog-Represäntation bei undefinerter Attributausprägung

```
1 student('Mayer', 21, u_n_d_e_f_i_n_e_d).
```

Das vorherige Beispiel würde dann wie im Listing 4.21 nach Prolog abgebildet werden. Die Abbildungskomponente muss dabei festhalten, an welcher Stelle welches Attribut steht. Soll nun ein Objekt der Klasse `Student` mit den Attributausprägungen `Name = Mayer` und `Alter = 21` abgebildet werden, so wird mittels eines speziellen Atoms vermerkt, dass im Unterschied zu den zuvor bekannten Objekten ein Attribut (nämlich das Studienfach) fehlt, siehe Listing 4.22. Soll nun ein Objekt der Klasse `Student` mit den Attributausprägungen `Name = Mueller`, `Alter = 22` und `maenlich = True` abgebildet werden, so muss die Abbildungskomponente ihr Konzept der Klasse `Student` um das Attribut `maenlich` erweitern, wie in Listing 4.23. Damit nun auch die bisherigen Atome mit dem neuen Konzept verarbeitet werden können, müssen zusätzliche Regeln, wie in Listing 4.24, zugesichert werden.

In der Implementierung von CAPPy werden zwei Möglichkeiten genutzt. Einerseits bietet es eine dynamische Abbildungskomponente an, die bei veränderter Attributmenge wie im letzten Punkt der Lösungsvorschläge verfährt. Anderseits bietet es auch statische Abbildungsverfahren an, bei denen der Nutzer die Attributmenge festlegt und die Abbildungsmechanismen beeinflussen kann. Die Abfragen werden in CAPPy stets erst zur Laufzeit übersetzt.

Dynamische Typisierung. Eine Sprache ist statisch typisiert, wenn die Typbedingungen zur Übersetzungszeit überprüft werden. Eine Sprache ist dynamisch typisiert, wenn die Typbedingungen zur Laufzeit überprüft werden. In der Praxis besitzen nahezu alle höheren Programmiersprachen beide Typisierungen (Vgl. [GM10, S.202f]). Java ist weitgehend statisch typisiert und daher kann im Programmcode in der Regel abgelesen werden, welche Typzuweisung ein Attribut be-

Listing 4.23: Erweiterte Prolog-Represäntation

```
1 student('Mueller', 22, u_n_d_e_f_i_n_e_d, true)
```

Listing 4.24: Überführungsregeln der PROLOG-Repräsentationen

```
1 student(A, B, C) :- student(A, B, C, _)
2 student(A, B, C, u_n_d_e_f_i_n_e_d) :-
3  ↪ student(A, B, C)
```

sitzt. Eine Ausnahme hiervon sind die generischen JAVA-Typen. Bei ihnen steht erst zur Laufzeit fest, welchem Typ sie angehören. PYTHON ist eine dynamisch typisierte Programmiersprache. Es steht daher in den meisten Fällen erst zur Laufzeit fest, zu welchem Typ ein Wert gehört.

In JAVA muss jedes Attribut deklariert werden. Das bedeutet, dass festgelegt wird, welchem Typ ein in diesem Attribut gespeicherter Wert entsprechen muss. Daher kann durch die Analyse des JAVA-Quellcodes eine Abbildungsklasse erstellt werden, in der die Konvertierungsvorschriften genau festgelegt sind. In PYTHON kann jedoch in jeder Variable und damit auch in jedem Attribut ein beliebiger Wert, unabhängig von seinem Typ, enthalten sein. Hierfür können folgende Lösungsvorschläge für die Implementierung von CAPPy unterbreitet werden:

- Die Benutzung von CAPPy wird darauf eingeschränkt, dass eine Attributsausprägung nur ein Wert eines bestimmten Typs enthalten darf. Die Information über diesen Typ wird mittels Annotationen CAPPy zur Verfügung gestellt, womit es eine Abbildungsklasse für diesen Typ erstellen kann.
- Die Typen der Attributsausprägungen werden dynamisch zur Laufzeit individuell analysiert und dem Typ entsprechend abgebildet. Der Vorteil bei dieser Methode besteht darin, dass keine natürliche Funktionalität aus PYTHON weggenommen wird, weil der PYTHON-Nutzer wie gewohnt die dynamische Typisierung ohne Einschränkung nutzen kann. Die zusätzliche Analyse des Typs verlangsamt allerdings die Verarbeitung.

Bei einem statischen Abbildungsverfahren wird der erste Lösungsvorschlag eingesetzt und bei der dynamischen Abbildungskomponente der zweite. Auf die Übersetzungskomponente hat die dynamische Typisierung keinen Einfluss.

Interpretierung und Kompilierung. Ein JAVA-Programm muss vor seiner Ausführung stets kompiliert werden. CAPJA analysiert den JAVA-Quellcode, um zum Beispiel Abbildungsklassen zu erstellen oder Abfragen an PROLOG zu parsen.

Ein PYTHON-Programm kann direkt einem PYTHON-Interpreter übergeben werden, der dieses dann ausführt. Dies stellt eine essentielle Eigenschaft von PYTHON

dar. Ein PYTHON-Programm kann auch kompiliert werden, wodurch es schneller ausgeführt werden kann.

Die Implementierung von CAPPy beachtet dies, indem es dynamische Komponenten für das Abbilden der Strukturen und das Übersetzen der Abfragen anbietet. Zusätzlich ermöglicht es eine statische Integration der Strukturen, durch die die Performance gesteigert werden kann.

Datenkapselung. In JAVA herrscht das Prinzip der Datenkapslung. Es dient dazu, dass der Programmierer festlegen kann, welche Attribute eines Objektes von den Objekten anderer Klassen ansprechbar sind. Dabei ist es speziell in JAVA üblich auch solche Variablen als `private` zu deklarieren, die von außen aus zugreifbar sein sollen, wobei der Zugriff dann durch Methoden ermöglicht wird, die selbst als `public` deklariert sind. Diese Methoden stellen damit eine Schnittstelle für den Zugriff von Objekten anderer Klassen dar und helfen so ungültige Wertzuweisungen zu verhindern (Vgl. [Ull, Kap. 5.2.1, 5.2.3 und 5.2.5]).

In PYTHON hingegen existiert keine solche Möglichkeit für die Datenkapslung. Es gibt daher insbesondere keine `privaten` Variableninstanzen. Um dennoch anzuzeigen, dass bestimmte Variablen nur innerhalb einer Klasse modifiziert werden sollen, hat sich eine Konvention etabliert. So soll auf ein Attribut nur innerhalb einer Klasse zugegriffen werden, wenn das erste Zeichen des Attributbezeichners ein Unterstich ist. Hiervon gibt es eine Ausnahme: Wenn ein Name bereits aus mindestens zwei führenden und höchstens einem folgendem Unterstich besteht, dann wird als Präfix ein Unterstrich, gefolgt von dem Klassennamen, hinzugefügt (Vgl. Name Mangling in [vPc, S. 76-77]).

Bei CAPJA kann der Nutzer mithilfe der JAVA *Beans* (also den Methoden die den Zugriff auf die Attribute erlauben) definieren, welche Werte nach PROLOG abgebildet werde sollen. Da in PYTHON ein solches Konzept unüblich ist, kann CAPPy an ihnen auch nicht feststellen, welche Attribute abgebildet werden sollen. Der Nutzer könnte aber die Namenskonventionen für quasi-private Attribute nutzen, um CAPPy auf natürliche Art mitzuteilen, welche Attribute abzubilden sind und welche nicht.

Dennoch wird in der Implementierung von CAPPy darauf verzichtet. Sollen nicht alle Attribute abgebildet werden, so kann dies über eine spezielle Methode angezeigt werden. Diese erbt die Klasse von einer bestimmten Oberklasse. Auf die Übersetzungskomponente hat die Datenkapselung keine Auswirkungen.

Vergleichsoperatoren. Vergleichsoperatoren vergleichen Ausdrücke miteinander und ergeben einen booleschen Wahrheitswert. In Java können die Operatoren Größer (>), Kleiner (<), Größer-gleich (>=) und Kleiner-gleich (<=) nur für numerische Vergleiche benutzt werden (Vgl. [Ull, Kap. 2.4.6]). In Python können diese Operatoren die Werte zweier beliebiger Typen vergleichen. Die allgemeinere Syntax für Vergleichsoperatoren erleichtert das Entwickeln der internen DSL für CAPPy, wie z. B. der Ppql.

Konsequenzen der Unterschiede für CAPPy

Als zentraler Unterschied zwischen CAPPy und CAPJa zeigt sich, dass CAPPy, im Gegensatz zu CAPJa, auch eine dynamische Verarbeitung ermöglichen muss. Die Quellcodeanalyse und -generierung ist bei CAPPy daher nur ein Weg es zu nutzen. Der Entwickler kann sie einsetzen, wenn er sein Programm hinsichtlich der Verarbeitungsgeschwindigkeit optimieren will. Dabei kann er nicht wie gewohnt die dynamischen Eigenschaften von Python nutzen und muss den Quellcode ausreichend annotieren. Als Alternative bietet CAPPy eine dynamische Verarbeitungskomponente an. Sie entscheidet zur Laufzeit *was wie* abgebildet wird. Wenn sich das Muster ändert, dann verändert sie die Abbildungsstrategie. Insgesamt können nicht beliebige Python-Typen abgebildet werden, sondern nur Objekte. Die Abfragen werden in CAPPy erst zur Laufzeit übersetzt.

4.2.4 Werkzeuge und Spracheigenschaften

In CAPJa werden eine Reihe von Werkzeugen bzw. Sprachfeatures eingesetzt. Hinsichtlich der Verfügbarkeit der in CAPJa genutzten Werkzeuge und Spracheigenschaften wird in diesem Abschnitt festgestellt, dass der Implementierung von CAPPy keine Hürden im Weg stehen, da diese Technologien auch für bzw. in Python existieren.

Antlr. Nach Parr [Par13, S. IX-X] ist Antlr ist ein mächtiger und verbreiterter Parser-Generator, der strukturierte Text- und Binärdaten lesen, ausführen und übersetzen kann. Er kann genutzt werden, um verschiedene Arten von Werkzeugen zu erschaffen, wie Code-Konverter, Extraktionsprogramme für bestimmten Code oder Datenbankabbildungen für objektrelationale Programme.

Eine Grammatik ist eine formale Beschreibung einer Sprache. Antlr generiert aus ihr einen Parser für diese Sprache, der automatisch Syntaxbäume erzeugen

Listing 4.25: Annoynme Funktion in Python

```
1 lambda [arguments]: expression
```

kann. Das sind Datenstrukturen, die angeben, wie eine Grammatik eine Eingabe verarbeitet. Dazu wird auch ein Treeworker generiert, mit dem die Knoten des Syntaxbaumes abgelaufen werden können. Antlr ist auch für Python erhältlich.

Antlr wird zum Beispiel in der Übersetzungs-Komponente von CAPPy eingesetzt, wenn eine Ppql-Abfrage in eine äquivalente Prolog-Abfrage umgeformt werden soll. Dazu analysiert es die Abfrage mittels Antlr. Da es sich um syntaktisch korrekten Python-Code handelt, kann eine bereits existierende Antlr-Grammatik für Python verwendet werden und so ein Syntaxbaum erstellt werden. Durch Ablaufen dieses Syntaxbaumes wird dann die korrespondierende Abfrage abgeleitet, in dem für jede Struktur in Ppql eine korrespondierende Prolog-Struktur erstellt wird.

Reflexion. Nach Malenfant et al. [MJD96, S. 2] bedeutet Reflexion die integrale Fähigkeit eines Programms seinen eigenen Code und alle Eigenschaften der Programmiersprache auch zur Laufzeit zu beobachten und zu verändern. Eine Programmiersprache ist reflexiv, wenn ihren Programme die Reflexion zugänglich ist. Mit *integral* ist an dieser Stelle gemeint, dass bei echter Reflexion keinerlei Einschränkungen gäbe, was das Programm beobachten bzw. modifizieren kann. Dies kann nie vollständig erreicht werden, denn es gibt hierbei auch eine Reihe von theoretischen Einschränkungen (Gödelscher Unvollständigkeitssatz, pardoxe Situationen etc.). Die Reflexion muss daher nur möglichst weitgehend ermöglicht werden. Python ist eine reflexive Programmiersprache.

Anonyme Funktionen. Da die Programmiersprache Python auch funktionale Programmierelemente kennt, ermöglicht es auch anonyme Funktionen, die auch Lambda-Ausdrücke genannt werden. Das sind Funktionen, die keinen Bezeichner besitzen. In Python [vPb, S.117] besteht ein Lambda-Ausdruck aus einem einzigen Ausdruck, der aufgerufen wird, wenn die Funktion ausgewertet wird. Seine Syntax wird in Listing 4.25 definiert. `[arguments]` ist eine Liste der Argumente der Funktion. `expression` ist der Ausdruck, der ausgewertet wird, wenn die Funktion aufrufen wird.

CAPJa verwendet anonymen Funktionen in der Abfragesprache Jpql in Form einer internen DSL. CAPPy verwendet anders als CAPJa Python-Funktionen

für die Abfragespezifikation. Der Grund hierfür ist, dass in Python in anonymen Funktionen keine Typhinweise für die Parameter erlaubt sind. In Python können auch Funktionen innerhalb von anderen Funktionen definiert und Funktionen so verschachtelt werden. Die zusätzliche Funktionsdefinition stört die restliche Programmstruktur folglich nicht. Der Nachteil bei diesem Ansatz ist, dass die Abfragen länger werden, weil sie immer benannt werden müssen. Andererseits sind die Namen ein Vorteil, weil der Programmcode so leichter verständlich wird und auf die Abfragen referenziert werden kann.

4.2.5 Implementierung eines Prototyps

Um die praktische Umsetzbarkeit zu zeigen, wurde ein Prototyp programmiert. Die korrespondierende Strukturen wurden analog zum Diagramm in Abbildung 4.1 und Listing 4.2 als Universitäts-Datenbank in Prolog und Python implementiert. Dazu wurden manuell Abbildungsklassen erstellt. Diese können die entsprechenden Objekte und Terme, wie zuvor beschrieben, zwischen Prolog und Python abbilden. Als Gateway wurde PySWIP genutzt. Aufgrund seiner funktionellen Einschränkungen konnten nur vordefinierte Abfragen ausgeführt werden. Dies war aber bereits ausreichend, um zu zeigen, dass der Prototyp funktionsfähig ist. Insbesondere können mit ihm Strukturen zwischen Prolog und Python ohne Informationsverlust hin und her übertragen werden. Auch wenn viele Funktionen CAPPy nicht enthalten sind, ist dieser Prototyp ein Indiz dafür, dass sich die Konzepte von CAPPy prinzipiell umsetzen lassen.

4.2.6 Ergebnis der Machbarkeitsstudie

In dieser Machbarkeitsstudie wurde gezeigt, dass das Werkzeug CAPPy realisierbar ist. Als Vorlage benutzt es dafür CAPJa, ein objektorientiertes Integrationsframework für Prolog und Java. Damit die Konzepte von CAPJa auf CAPPy übertragen werden können, wurden die sprachlichen Unterschiede von Python und Java herausgearbeitet. Insbesondere unterschieden sich Python von Java, indem es wesentlich dynamischer ist und seine Programme auch ohne Kompilierung lauffähig sind. Deswegen bietet CAPPy auch eine dynamische Abbildungskomponente und einen dynamischen Abfrageparser an, die ohne Vorab-Analyse des Quellcodes funktionsfähig sind.

Weiterhin wurde ein Designvorschlag gemacht, der das zuvor abstrakte Konzept auf seine funktionellen Eigenschaften herunterbricht. Korrespondierende Strukturen in Prolog und Python ermöglichen ein intuitives Verständnis einer Pro-

log-Datenbank in Python. Die Abfragesprache Ppql ermöglicht es deklarativ, kompakt und intuitiv mit Prolog zu operieren, ohne hierfür Vorkenntnisse besitzen zu müssen. Die Abfragen in Ppql werden von CAPPy automatisiert in äquivalente Prolog-Abfragen übersetzt und an ein Gatway weitergeleitet, das prinzipiell alle Prolog-Interpreter unterstützen kann, aber zunächst nur für SWI-Prolog implementiert wird.

Die Automatisierungen erleichtern die Arbeit des Python-Programmierers erheblich. Insbesondere muss er die korrespondierende Struktur nicht selber erstellen und sich keine oder nur geringe Gedanken über die Abbildungsprozesse machen.

Auch Rdf-Daten lassen sich durch CAPPy leicht verarbeiten. Dazu wird eine Anbindung an ClioPatria etabliert, wobei die komplette Kommunikation über CAPPy läuft. Durch minimale Modifikationen an der Ppql, können auch Rdf-Daten einfach abgefragt werden, ohne dazu eine neue Sprache wie Sparql erlernen zu müssen.

5 Implementierung des Integrationsframeworks CAPPy

In diesem Kapitel wird die Implementierung von CAPPy erläutert. Hierzu werden zunächst die Designziele und der grundsätzliche Aufbau von CAPPy dargestellt. Anschließend wird die Implementierung und das strukturelle Design der drei Kernkomponenten PPGateway, PPMapping und PPQuery skizziert, die für die Verbindung zu einem Prolog-Interpreter, dem Abbilden der Strukturen und dem Auswerten von Ppql-Abfragen zuständig sind.

5.1 Einführung in CAPPy

Als ein Hauptbestandteil dieser Arbeit wurde das Tool CAPPy entwickelt. Es ist ein Integrationsframwork für Prolog in Python. Das elementare Ziel von CAPPy ist es, eine einfache und intuitiv bedienbare Schnittstelle anzubieten, die mit den aktuellen Versionen von Swi-Prolog und Python kompatibel ist. Im Folgenden wird eine Übersicht hierüber gegeben.

Konzeptioneller Aufbau. CAPPy besteht analog zu CAPJa [Ost17] aus drei Basiskomponenten, die den wesentlichen Teil der Implementierung ausmachen:

PPGateway Dies ist ein Gateway mit dem in Python prinzipiell beliebige Prolog-Interpreter angesprochen werden können. Deswegen sind nur solche Schnittstellen zu Prolog verfügbar, die Iso-Prolog-konform sind. PPGateway automatisiert die komplette Interaktion mit dem Interpreter und wandelt seine Ergebnisse in einfache Prolog-Strukturen um.

PPMapping Diese Komponente kann Prolog-Terme und Python-Objekte aufeinander abbilden. Dazu stellt die Komponente Abbildungsmechanismen bereit, die die Korrespondenz zwischen Prolog-Prädikaten und Python-Klassen definieren. Diese Abbildungsmechanismen können vollständig auto-

matisiert zur Laufzeit eingesetzt oder vorab statisch erzeugt werden, wobei im letzteren Fall der Abbildungsmechanismus angepasst werden kann.

PPQuery Dies ist die Abfragekomponente von von CAPPy. Mit ihr können objektorientierte Abfragen in Ppql an Prolog gestellt werden. Dazu wandelt PPQuery die Abfrage automatisiert in eine äquivalente Prolog-Abfrage um, wobei es sich auf die Abbildungsmechanismen von PPMapping stützt. Dadurch kann der Benutzer Prolog-Abfragen einsetzten und dabei nur mit Python-typischen Strukturen interagieren. Als zusätzliches Feature stellt es auch eine besondere Unterstützung für Rdf-Abfragen bereit.

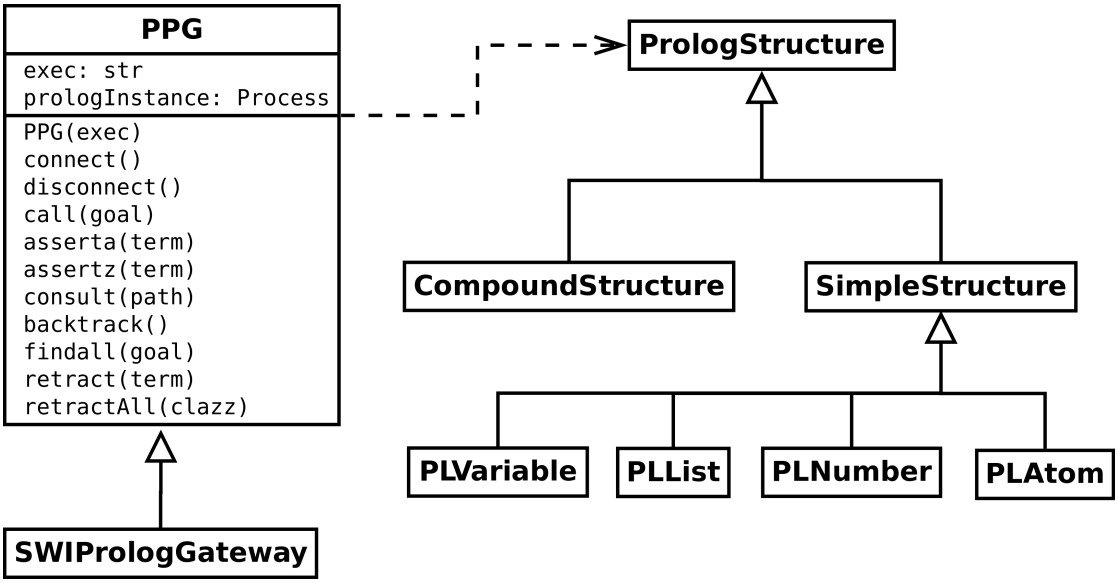
Im restlichen Teil dieses Kapitels wird die Implementierung dieser Basiskomponenten dargelegt.

5.2 Die Verbindungskomponente PPGateway

PPGateway steht für Prolog-Python-Gateway und ist ein System aus Gateways, das prinzipiell mit beliebigen Prolog-Interpretern kommunizieren kann, weil es nur eine Iso-Prolog-konforme Syntax benutzt. In dieser Arbeit wurde lediglich ein Gateway für Swi-Prolog implementiert, denn es wird von vielen Prolog-Anwendungen unterstützt – insbesondere von ClioPatria. PPGateway kann problemlos durch weitere Gateways ergänzt werden. Ein wichtiges Ziel dieser Komponente ist es, zumindest eine bessere Anbindung zu ermöglichen als in PySWIP. Im Folgenden wird auf das Design und die Implementierung von PPGateway detaillierter eingegangen.

Desing. In Abbildung 5.1 wird das Klassendesign von PPGateway vereinfacht dargestellt. Ppg steht für Portable Prolog Gateway wurde aus CAPJa [Ost17, S. 125-126] übernommen. Es dient in CAPPy als Abstraktion für die Gateways zu den einzelnen Prolog-Interpretern. Mit der Methode `connect` bzw. `disconnect` kann eine Verbindung zu einem Interpreter aufgebaut bzw. beendet werden. Durch die Methode `call` kann ein Goal in Prolog aufgerufen werden, in dem man es ihr als Parameter übergibt. Wenn eine weitere Lösung ausgegeben werden soll, kann die Methode `backtrack` verwendet werden. Sollen alle Lösungen gefunden werden, die auf das Goal zutreffen, bietet sich die Methode `findall` an. Die Methode `consult` lädt eine Prolog-Datei, deren Pfad in Form einer Zeichenkette als Parameter übergeben wird. Durch `asserta`, `assertz` und `retract` kann ein Term zugesichert bzw. gelöscht werden. Durch die Methode `retractAll` kön-

Abbildung 5.1: Klassenstruktur von PPGateway (vereinfacht)



nen alle Terme eines Prädikats gelöscht werden. Der Übergabeparameter ist hierbei eine korrespondierende Klasse in Python, die dieses Prädikat repräsentiert.

5.2.1 Implementierung des SwiPrologGateways

In diesem Abschnitt wird die Implementierung der Klasse SwiPrologGateway eingeführt. Dazu werden zunächst die eingesetzten Werkzeuge erläutert und anschließend die Verarbeitung einer Abfrage skizziert.

Antlr. Um die Variablen aus einer Prolog-Abfrage zu extrahieren, benutzt PPGateway eine Antlr-Grammatik. Als Grundlage für die eingesetzte Antlr-Grammatik dient eine Grammatik, die aus dem CAPJa-Projekt übernommen wurde und Prolog-Terme beschreibt. Damit auch vollständige Abfragen geparst werden können, musste sie noch um eine Beschreibung der *Build-In*-Prädikate erweitert werden.

PPGateway erstellt aus der Eingabe und dem von Antlr generierten *Parser* einen Syntaxbaum. Ein Objekt läuft diesen Syntaxbaum ab, weswegen er *Walker* genannt wird. Immer wenn er eine bestimmte Regel betritt oder verlässt, wird ein *Listener* aufgerufen. In diesem Fall wird der *Listener* nur aktiv, wenn eine Variable

verlassen wird. Der *Listener* speichert die gefunden Variablen in einer Liste ab. Nachdem der *Walker* den Baum vollständig abgelaufen ist, existiert somit eine vollständige Liste der Variablen aus der Abfrage.

Pexpect. PEXPECT ist ein PYTHON-Modul, mit dem Unterprogramme aufgerufen und kontrolliert werden können, als wenn ein Mensch ein Terminal-Programm bedienen würde. Auf die Ausgabe kann geantwortet werden, wenn ein zuvor spezifiziertes Muster in der Ausgabe aufgetreten ist (Vgl. [pex]). Das SWIPROLOG-GATEWAY nutzt PEXPECT, um mit dem PROLOG-Interpreter zu kommunizieren.

Implementierung. Um die Ergebnisse des PROLOG-Interpreters zu verarbeiten, nutzt SWIPROLOGGATEWAY einen Zustandsautomaten. Dies begründet sich mit dem Umstand, dass erkannt werden muss, an welcher Position der Ausgabe das Ergebnis vollendet ist. Der Automat liest daher die Ausgabe Zeichen für Zeichen ein und beginnt zu diesem Zeitpunkt bereits damit, die Ausgabe in PYTHON-Strukturen umzuwandeln. Diese Strukturen sind Objekte. In Abbildung 3.4.2 sind die Klassen der Objekte darstellt. Für jede der PROLOG-Typen *Compound* (gemeint ist das Prädikat), *Variable*, *List*, *Number* und *Atom* existiert eine entsprechende Klasse.

Vereinfacht gesagt, erwartet SWIPROLOGGATEWAY als mögliches Ergebnis `true`, `false`, eine Zuweisungen zu Variablen oder eine Fehlermeldung. Mit den aus der Abfrage extrahierten Variablen und den möglichen Ausgaben `true` oder `false` wird ein Muster erstellt, auf das gewartet wird. Weicht die Ausgabe davon ab, so wird dies als Fehlermeldung interpretiert.

Für jedes Zeichen der Ausgabe existiert eine Methode, die die Verarbeitung übernimmt. Die Methode liest sukzessive weitere Zeichen ein, bis eine Instanz vollständig eingelesen ist. Das ist möglich, weil in PROLOG aus der Syntax der Typ abgleitet werden kann. Daraus generiert die Methode ein entsprechendes PYTHON-Objekt. Die Interpretation wird anschließend rekursiv aufgerufen, um die weiteren Instanzen einzulesen.

Sei als Beispiel die PROLOG-Datenbank für Mitarbeiter aus Listing 5.1 gegeben. Eine PROLOG-Abfrage an diese Datenbank kann wie in Listing 5.2 an eine Instanz von SWIPROLOGGATEWAYs gestellt werden. Das Gateway extrahiert aus der Abfrage die Variable `X`. Es erwartet daher als gültigen Rückgabewert `true`, `false` oder eine Belegung der Variablen `X`. Andere Ausgaben würde es als Fehlermeldung interpretieren. Die unmittelbare Ausgabe des SWI-PROLOG-Interpreters, die es empfängt, kann in Listing 5.3 eingesehen werden. Da die Rückgabe des In-

Listing 5.1: Mitarbeiter-Datenbank in Prolog

```
employee('Max', 1234, address(10000, 'Musterstadt')).
employee('Peter', 45678, address(20000, 'Musterdorf')).
```

Listing 5.2: Einfache Abfrage an SwiPrologGateway

```
res = mySWIPrologGateway.call("employee(_, 1234, X)")
```

terpreters mit `X` anfängt, erkennt SwiPrologGateway, dass das Ergebnis eine Belegung der Variablen `X` ist. Es liest die restlichen Zeichen Schritt für Schritt ein, und erkennt an der Prolog-Syntax, was die Zeichen bedeuten und erzeugt entsprechende Python-Objekte. Das Ergebnis wird in Listing 5.4 dargestellt.

5.3 Die Abbildungskomponente PPMapping

PPMapping steht für Prolog-Python-Mapping und ist die Abbildungskomponente von CAPPy. Ihre Aufgabe ist es, die Prolog-Terme und die Python-Objekte aufeinander abzubilden. Dazu wird zunächst eine Abbildungsdefinition erstellt, die grundsätzlich festlegt, wie Prolog- und Python-Strukturen aufeinander abgebildet werden können. Damit es durch das Hin-und-Her-Abbilden zu keinen Veränderungen an den Daten kommt, ist die Abbildung bijektiv definiert. Anschließend wird erläutert, wie in PPMapping das Prinzip von dynamischer und statischer Abbildung funktioniert. Im letzten Abschnitt wird der schematische Aufbau von PPMapping dargestellt.

5.3.1 Abbildungsdefinition

Damit Prolog-Terme und Python-Objekte aufeinander abgebildet werden können, muss eine Abbildungsvorschrift definiert werden. Die Abbildungsvorschrift in PPMapping ist bijektiv. Das heißt, dass zu jedem Prolog-Term eine eindeutige

Listing 5.3: Ergebnis der Abfrage in Swi-Prolog

```
X = address(10000, 'Musterstadt').
```

Listing 5.4: Ergebnis der Abfrage in SwiPrologGateway

```
result = Assignment(PLVariable('X'),
 ↪  CompundStructure('address',
 ↪  [PLNumber(1000), PLAtom('Musterstadt')])
```

Objekt-Repräsentation in Python existiert und umgekehrt. Objekte und Terme können damit beliebig oft aufeinander abgebildet werden, ohne dass dadurch semantische Informationen verloren gingen. Im Folgenden wird die Definition der Abbildungsvorschrift detailliert dargelegt.

Jede Python-Klasse entspricht einem Prolog-Prädikat und umgekehrt, wobei der Klassenbezeichner dem Prädikatsbezeichner entspricht. Weil es in Python möglich ist, dass mehrere Klassen mit identischen Namen in unterschiedlichen Modulen existieren, wird immer der vollqualifizierte Klassenname verwendet, um etwaige Mehrdeutigkeiten aufzulösen.

Die Attribute einer Python-Klasse entsprechen den Komponenten eines Prolog-Prädikats. Da die Attribute in einer Python-Klasse durch ihre Bezeichner und die Komponenten eines Prolog-Prädikats durch ihre Stelligkeit angesprochen werden, muss für jede Abbildung eine eindeutiger Zusammenhang zwischen den Attributsbezeichnern und der Position der Komponenten bestimmt werden.

Ist eine Komponente eines Prolog-Prädikats wiederum ein Prolog-Prädikat bzw. ein Attribut eines Python-Objektes wiederum ein Python-Objekt, so ist die Abbildung in PPMapping rekursiv definiert. In allen anderen Fällen handelt es sich um *Build-In-Typen*, für die eine eigene Abbildungsvorschrift definiert wurde.

Abbildung der *Build-In-Typen*. Der Glanzzahlen-Typ `int` [vPb, S. 18-19] in Python kann auf die Prolog-Struktur `integer` abgebildet werden und der Fließkommazahl-Typ `float` [vPb, S. 18-19] auf die Prolog-Struktur `float`. Der Zahlentyp `complex` [vPb, S. 18-19], wird auf eine neue Prolog-Struktur `complex(r, i)` abgebildet, wobei `r` den reellen Teil und `i` den imaginären Teil beschreibt. Es wird eine neue Prolog-Struktur definiert, anstatt eine zu verwenden, die in einem gegeben Prolog-Interpreter vordefiniert ist. Dadurch ist die Repräsentation Iso-Prolog-kompatibel.

Der Python-Typ für Zeichenketten `str` [vPb, S. 19] kann auf Prolog-Atome abgebildet werden und umgekehrt. Bei der Abbildung von Prolog-Atomen zu Python-Strings gibt es keine besonderen Probleme, da Python Unicode voll

unterstützt. Im umgekehrten Fall, können aber Sonderzeichen Probleme bereiten, weil nicht jeder PROLOG-Interpreter notwendigerweise UNICODE-Zeichen unterstützt. Alle Buchstaben von *A* bis *Z* (je in Groß- und Kleinschreibung) und Zahlen werden direkt abgebildet. Die restlichen Zeichen werden durch die Hexadezimal-Repräsentation der UNICODE-Werte dargestellt. Zur Markierung wird ein Unterstrich voran- und nachgestellt. Da keine PROLOG-Atome mit Großbuchstaben beginnen dürfen, wird der String noch mit einfachen Hochkommata umschlossen. So wird beispielsweise der PYTHON-String `Börsenstraße` zu dem PROLOG-Atom `'B_f6_rsenstra_df_e'` abgebildet. Diese PROLOG-Repräsentation mag ungewöhnlich sein. Sie ist aber eindeutig und mit allen PROLOG-Interpretern kompatibel. Da die abgebildeten Terme vornehmlich von CAPPy verarbeitet werden, kann diese Repräsentation als gerechtfertigt betrachtet werden. Wenn ein PROLOG-Atom zu einem PYTHON-String abgebildet werden soll, beachtet PPMAPPING dabei, ob es sich bei der Instanz um ein natives PROLOG-Atom oder eine Abbildung eines PYTHON-Strings handelt und nimmt die Sonderzeichen-Behandlung nur in letzterem Fall vor. In gewisser Weise ist die Abbildung an dieser Stelle nicht mehr bijektiv.

Die PROLOG-Struktur `list` kann direkt auf den PYTHON-Typ `list` abgebildet werden, da sie beide Container-Typen sind, die ihre Werte geordnet enthalten und mehrfaches Vorkommen erlauben. Die anderen Container-Typen in PYTHON `tuple`, `set` und `frozenset` könnten auch direkt auf PROLOG-Listen abgebildet werden. Dadurch wäre die Bijektivität der Abbildung nicht mehr gewahrt. Sie werden deshalb auf neue PROLOG-Prädikate abgebildet, deren einziges Element eine korrespondierenden PROLOG-Liste ist. Die Prädikate heißen entsprechend `tuple/1`, `set/1` und `frozenset/1`.

Der `range`-Typ in PYTHON ist eine Struktur, die eine Reihe von natürlichen Zahlen repräsentiert. Eine Instanz des `range`-Typs ist definiert durch den Startwert `start`, dem Stoppwert `stop` und der Schrittweite `step`. Dieser Typ wird auf das neue PROLOG-Prädikat `range(start, stop, step)` abgebildet.

Die PYTHON-Typen `bytes`, `bytearray` und `memoryview` dienen zur Repräsentation von Binärdaten. Sie werden auf die PROLOG-Prädikate `bytes/1`, `bytearray/1` und `memoryview/1` abgebildet und enthalten als alleiniges Element ein Atom in Form der Hexadezimaldarstellung der Binärdaten.

Ein assoziatives Datenfeld wird in PYTHON durch den Typ `dict` realisiert. Es ist somit eine Ansammlung von Schlüssel-Wert-Paaren. Es wird auf das neue PROLOG-Prädikat `dict(keys, values)` abgebildet. Dabei ist `keys` eine PROLOG-Liste, die die Schlüssel enthält und `values` eine PROLOG-Liste, die die Werte

Tabelle 5.1: Bijektive Abbildungsvorschrift der *Build-In-Typen*

PYTHON	PROLOG
int	integer
float	float
complex	complex(r, i)
str	atom
list	PROLOG-Liste
tuple	tuple(PROLOG-Liste)
set	set(PROLOG-Liste)
frozenset	frozenset(PROLOG-Liste)
range	range(start, stop, step)
byte	byte(*Hexadezimaldarstellung*)
bytearray	bytearray(*Hexadezimaldarstellung*)
memoryview	memoryview(*Hexadezimaldarstellung*)
dict	dict(keys, values)
True	true
False	false
None	n_o_n_e
NotImplemented	n_o_t_i_m_p_l_e_m_e_n_t_e_d
Ellipsis	e_l_l_i_p_s_i_s

enthält. Dabei ist das i-te Element in `keys` der Schlüssel des i-ten Elements in `values`.

Die statischen Objekte in PYTHON werden auf PROLOG-Atome abgebildet. Die booleschen Werte werden auf ihre jeweiligen Äquivalente abgebildet. Die anderen statischen Objekte `NotImplemented`, `Ellipsis` und `None`, werden auf die PROLOG-Atome `n_o_t_i_m_p_l_e_m_e_n_t_e_d`, `e_l_l_i_p_s_i_s` und `n_o_n_e` abgebildet.

In PPMAPPING wird davon ausgegangen, dass die neuen PROLOG-Prädikate und die speziell definierten PROLOG-Atome in keiner angebunden Datenbank bereits existieren. In der Tabelle 5.1 werden alle Abbildungsvorschriften der *Build-In*-Typen übersichtlich dargestellt.

5.3.2 Dynamische und statische Abbildung

PPMAPPING eröffnet zwei Möglichkeiten der Abbildung von PROLOG-Prädikaten und PYTHON-Klassen und zwar die statische und dynamische Abbildung. Bei der

statischen Abbildung werden Abbildungsklassen und korrespondierende Strukturen erzeugt, die fest in den Programmcode eingefügt werden. Der Abbildungsprozess kann insbesondere vom Programmierer angepasst werden. Da PYTHON-Module angelegt werden, die kompiliert werden können, kann so die Verarbeitungsgeschwindigkeit von CAPPy verbessert werden. Die dynamische Abbildung dient primär der direkten und schnellen Integration von PROLOG und PYTHON. Sie kann insbesondere ohne Vorabverarbeitung eingesetzt werden und ist mit beliebigen PROLOG- und PYTHON-Code kompatibel. Sämtliche Abbildungsprozesse finden nach einer nicht-anpassbaren Vorschrift statt und werden nur teilweise in den Programmcode eingebunden. Diese Komponente trägt dadurch der PYTHON-typischen Dynamik Rechnung. Die Begriffe dynamische und statische Abbildung beziehen sich dabei darauf, zu welchem Zweck die Abbildung angedacht ist und nicht darauf, wie die Abbildungsprozedur funktioniert. Durch die PYTHON-typische Dynamik können prinzipiell auch die statischen Abbildungen zur Laufzeit eingesetzt werden. Im Folgenden wird gezeigt, wie die statische und dynamische Abbildung in CAPPy implementiert wurde.

Dynamische Python-zu-Prolog-Abbildung. Die dynamische PYTHON-zu-PROLOG Abbildung basiert auf einem Prototypen-Prinzip. Wird PPMAPPING zum ersten Mal aufgerufen, ein Objekt einer gegebenen PYTHON-Klasse in ein PROLOG-Term zu überführen, so wird das PYTHON-Objekt als Prototyp hinsichtlich der Attributmenge für alle Objekte seiner Klasse betrachtet. Mittels Reflexion werden die Attributsbezeichner aus dem Objekt extrahiert. Für diese Menge wird eine CAPPy-interne Reihenfolge festgelegt und gespeichert. Wird bei einer neuen dynamischen Abbildung eines PYTHON-Objekts festgestellt, dass sich die Attributmenge von der gespeicherten unterscheidet, wird wie in Kapitel 4.2.3, Abschnitt *Dynamische Erweiterbarkeit*, letzter Unterpunkt dargestellt, das Konzept erweitert.

Sobald feststeht an welcher Position im PROLOG-Prädikat welches Attribut des abzubildenden PYTHON-Objektes stehen soll, werden die Attribute des Objektes einzeln analysiert. Bei der dynamischen Abbildung können in den Attributen beliebige Typen verwendet werden. PPMAPPING analysiert welchem Typ die Attributsausprägung konkret entspricht und bildet sie entsprechend ab. Da jedes Attribut einzeln analysiert werden muss, erhöht dies die Verarbeitungszeit im Vergleich zu dem Fall, in dem der Typ bereits fest steht und auf die Analyse des Typs verzichtet werden kann. Der Benutzer kann das Abbildungsverhalten in der dynamischen Abbildung nicht beeinflussen.

Statische Python-zu-Prolog-Abbildung. Die statische PYTHON-zu-PROLOG Abbildung basiert auf Annotationen im Quellcode. Um diese Form der Abbildung einzusetzen, muss die abzubildende Klasse eine Unterklasse von `StaticallyMappable` sein. Diese definiert eine Methode `setAttributes`, die überschrieben werden muss und CAPPy so eine Spezifikation der gewünschten Abbildung zur Verfügung stellt. Im Gegensatz zur dynamischen Abbildung hat der Benutzer hier die Möglichkeit, die Attribute zu definieren, die abgebildet werden sollen. Wenn zum Beispiel Attribute eines Objekts in der PROLOG-Repräsentation unwichtig sind, so können sie in der Signatur einfach weggelassen werden und dadurch die PROLOG-Repräsentation schlanker gestalten. Weiterhin muss der Benutzer jedem Attribut einen festen Typ zuordnen. Die statische Abbildung stellt damit eine Einschränkung der PYTHON-typischen Dynamik dar.

Aus der Signatur der Methode `setAttributes` folgt die Abbildungsvorschrift. Die Namen der Parameter definieren die abzubildenden Attribute, samt Reihenfolge und Typ. Im Methodenrumpf sollen die Attribute gesetzt werden. Hierbei hat der Programmierer freien Spielraum. Er kann auch Werte für weitere Attribute setzten, die nicht in der Signatur erfasst sind.

Die Funktionalität der Methode `setAttributes` ähnelt offensichtlich der eines Konstruktors. Dennoch wurde in PPMAPPING darauf verzichtet, den Konstruktor als Definition der Abbildungsvorschrift zu verwenden, um dem Programmierer möglichst viel Freiraum zu lassen. So kann es sein, dass nicht alle Attribute, die im Konstruktor auftauchen, auch abgebildet werden sollen. Umgekehrt können auch Attribute nicht im Konstruktor auftauchen, die abgebildet werden sollen. Da PPMAPPING den Konstruktor beim Erzeugen neuer Objekte umgeht und die Attribute durch die `setAttributes` setzt, kann der Programmierer dies ausnutzen und Objekte, die auf herkömmlichen Wege durch den Konstruktor erzeugt werden, und solche, die von PPMAPPING erzeugt werden, anders initialisieren. In vielen Fällen kann es aber sinnvoll sein, dass die Implementierung die selbe ist. In diesem Fall wird empfohlen im Konstruktor die zu setzenden Attribute an die Methode `setAttributes` weiterzuleiten.

Da die statische Abbildung auch auf eine Performancesteigerung abzielt, findet in den Abbildungsklassen keine Typprüfung statt. Stattdessen obliegt dem Benutzer die Aufgabe, nur Objekte, die der Abbildungsvorschrift entsprechen, abbilden zu lassen.

Sei als Beispiel die PYTHON-Klasse `Student` wie in Listing 5.5 gegeben. Aus der Signatur `setAttributes` geht hervor, dass in ihr die Attribute `first_name` mit dem Typ `str`, `last_name` mit dem Typ `str`, `age` mit dem Typ `int`, `male` mit

Listing 5.5: PYTHON-Klasse Student

```
from src.de.uni.wue.cappy.PPMapper.StaticallyMappable
 ↪ import StaticallyMappable
from ExampleStructures import Address

class Student(StaticallyMappable):
 def __init__(self, first_name, last_name, age, male
  ↪ account_balance, address):
   self.setAttributes(first_name, last_name, age, male
    ↪ account_balance, address)

 def setAttributes(self, first_name: int,
  ↪ last_name: str, age: int, male: bool,
  ↪ account_balance: float, address: Address):
   self.first_name = first_name
   self.last_name = last_name
   self.age = age
   self.male = male
   self.account_balance = account_balance
   self.address = address
```

dem Typ `bool`, `account_balance` mit dem Typ `float` und `address` mit dem Typ `Address` existieren und abgebildet werden sollen.

PPMAPPING liest die Signatur mittels Reflexion ein und erzeugt eine Abbildungsklasse. Die generierte Abbildungsklasse der Klasse `Student` findet sich vereinfacht in Listing 5.6. Für eine leichtere Lesbarkeit wurden die Modulnamen gekürzt. Um Mehrdeutigkeiten zu vermeiden, werden die Klassen in der eigentlichen Implementierung stets mit vollqualifizierten Namen bezeichnet.

Die Abbildungsklasse besitzt die Methoden `termToObject` und `objectToTerm`. Die erste Methode bildet einen PROLOG-Term auf ein PYTHON-Objekt ab und die zweite Methode bildet ein PYTHON-Objekt auf einen PROLOG-Term ab. Die Implementierung der Methoden folgt aus der Abbildungsprozedur der dynamischen Abbildung. Da der Code leicht zugänglich ist, können fortgeschritten CAPPY-Benutzer ihn modifizieren und so die Abbildungsvorschrift vollständig anpassen. Weiterhin enthält jede Abbildungsklasse eine Methode `getAttributeNameList`. Sie gibt eine Liste zurück, aus der die Positionen folgen, an der die Attribute eines PYTHON-Objektes in einen PROLOG-Term abgebildet werden und umgekehrt.

Des Weiteren können die Klassen noch zusätzliche Methoden beinhalten, die zur Abbildung von Zeichenketten, Listen etc. dienen. In diesem Fall existieren die Methoden `parsePrologString` und `parsePyString`. Sie dienen zur Konvertierung von Zeichenketten und wurden aus Gründen der Übersichtlichkeit im Listing 5.6 weggelassen.

Dynamische Prolog-zu-Python-Abbildung. Die dynamische PROLOG-zu-PYTHON Abbildung kann beliebige PROLOG-Terme auf PYTHON-Objekte abbilden, wobei die Prädikate der PROLOG-Terme nicht unbedingt annotiert sein müssen. Sie ist dann sinnvoll, wenn der Benutzer bereits von anderer Seite aus Kenntnis über die Struktur der PROLOG-Datenbank besitzt. Er kann dann die dynamische Abbildung verwenden, um Abfragen an die PROLOG-Datenbank in CAPPy zu generieren. Sie kann auch sinnvoll sein, wenn CAPPy ein Prädikat ausgeben will, für das noch keine Abbildung statisch definiert wurde. Der Fokus dieser Abbildungsmethode liegt primär darauf, überhaupt eine Abbildung von PROLOG nach PYTHON zu ermöglichen und weniger darauf, dass die Abbildungen leicht zugänglich sind.

PPMAPPING bedarf hierfür lediglich eines Beispiel-Terms des abzubildenden Prädikats. Es erzeugt daraus eine einfache PYTHON-Klasse. Da PROLOG nicht statisch typisiert ist und die Komponenten eines Prädikats über keine Bezeichner verfügen, können keine interessanten Informationen aus dem Beispielfakt extrahiert werden.

Sei als Beispiel der PROLOG-Term `employee/5` aus dem Listing 5.7 gegeben. PPMAPPING erzeugt aus diesem PROLOG-Term zwei PYTHON-Klassen – eine für das Prädikat `employee/5` und eine für das Prädikat `address/2`. Diese Klassen können dem Listing 5.8 entnommen werden.

Statische Prolog-zu-Python-Abbildung. Damit auch aussagekräftige Klassen generiert werden können, eröffnet CAPPy eine weitere Abbildungsmethode von PROLOG zu PYTHON, die statische Abbildung genannt wird. Sie zielt hauptsächlich auf einer produktiven Weiterverarbeitung der erzeugten Klassen ab. Dies ist aber nur unter der Einschränkung möglich, dass die PROLOG-Prädikate zuvor durch die PMN annotiert wurden.

Um die statische PROLOG-zu-PYTHON Abbildung durchzuführen, liest PPMAPPING die zu dem Prädikat gehörende PMN-Struktur ein. Hieraus erzeugt es eine korrespondierende Klasse.

Listing 5.6: Statische Abbildungsklasse für `Student` (vereinfacht)

```
from ExampleStructures import Student
from ExampleStructures import Address
from Mapper_ExampleStructures_Address
 ↪ import Mapper_ExampleStructures_Address
from AStaticMappingClass import AStaticMappingClass

class Mapper_ExampleStructures_Student(
 ↪ AStaticMappingClass):

 @staticmethod
 def termToObject(term):
  return Student(Mapper_ExampleStructures_Student.
   ↪ parsePrologString(term.value[0].value),
   ↪ Mapper_ExampleStructures_Student.
   ↪ parsePrologString(term.value[2].value),
   ↪ int(term.value[2].value),
   ↪ (True if term.value[3].value == 'true' else False),
   ↪ float(term.value[4].value),
   ↪ Mapper_ExampleStructures_Address.termToObject(term.
   ↪ value[5].value))

 @staticmethod
 def objectToTerm(obj):
  return '_ExampleStructures_Student(' +
   ↪ Mapper_ExampleStructures_Student.
   ↪ parsePyString(obj.last_name) + ', ' +
   ↪ Mapper_ExampleStructures_Student.
   ↪ parsePyString(obj.first_name) + ', ' +
   ↪ str(obj.age) + ', ' +
   ↪ ('true' if obj.sex else 'false') + ', ' +
   ↪ str(obj.account_balance) + ', ' +
   ↪ Mapper_ExampleStructures_Address.
   ↪ objectToTerm(obj.address) + ')'

 @staticmethod
 def getAttributeNameList():
  return ['last_name', 'first_name', 'age', 'sex',
   ↪ 'account_balance', 'address']
```

Listing 5.7: PROLOG-Term `employee/5`

```
employee('Meier', true, 12, 13.5,
 ↪ address('hauptstrasse', 1))
```

Listing 5.8: Dynamische Abbildungsklasse für `employee/5`

```
from src.de.uni.wue.cappy.PPMapper.AGenClass
 ↪ import AGenClass

class Predicate_employee_5(AGenClass):
 def __init__(self, pos0, pos1, pos2, pos3, pos4):
  self.pos0 = pos0
  self.pos1 = pos1
  self.pos2 = pos2
  self.pos3 = pos3
  self.pos4 = pos4

class Predicate_address_2(AGenClass):
 def __init__(self, pos0, pos1):
  self.pos0 = pos0
  self.pos1 = pos1
```

Listing 5.9: PMN-Strukturen für `address/2` und `employee/5`

```
predicate(address, [street, number], [str, int])
predicate(employee,
 ↪ [name, valid, dep_nr, overtime, address],
 ↪ [str, int, bool, float, address])
```

Seien als Beispiel die PMN-Strukturen für `address/2` und `employee/5`, wie im Listing 5.9 gegeben. Damit die PMN-Struktur für das Prädikat `employee/5` erstellt werden kann, muss zuerst die des Prädikats `address/2` erzeugt werden, weil in der PMN-Definition von `employee/5` auf das Prädikat `address/2` referenziert wird. Das vereinfachte Abbildungsergebnis für die PMN-Struktur `employee/5`, kann in Listing 5.10 eingesehen werden. Die generierte Klasse enthält Bezeichner und Typhinweise für alle Attribute. Einem PYTHON-Programmierer ist die Struktur des ursprünglichen PROLOG-Codes dadurch sofort klar. Eine PYTHON-IDE kann aus diesem Code auch die Attribute extrahieren und so zusätzliche Funktionalitäten, wie Autovervollständigung, dem Benutzer anbieten und dadurch die Benutzerfreundlichkeit verbessern.

Für die erzeugte Klasse generiert PPMAPPING auch noch eine statische Abbildungsklasse. Da aus der PMN-Struktur eine valide PYTHON-Klasse generiert wurde, die auch die strukturellen Voraussetzungen für eine statische PYTHON-zu-PROLOG-Abbildung beinhaltet, wird die statische PYTHON-zu-PROLOG-Abbildung verwendet, um die Abbildungsklasse zu generieren.

5.3.3 Schematischer Aufbau von PPMapping

In diesem Abschnitt wird der schematische Aufbau von PPMAPPING eingeführt. Dazu wird das Klassendesign erläutert und die Verwaltung der Abbildungsvorschriften dargelegt.

Das Klassendiagramm der PPMAPPING-Komponente kann aus der Abbildung 5.2 entnommen werden. Der Großteil der Verarbeitungslogik befindet sich in der Klasse `Mapper`. Ihre Methoden `pythonObjToPrologTerm` und `prologTermToPythonObj` können PYTHON-Objekte in PROLOG-Terme abbilden und umgekehrt. Sie prüfen, ob bereits eine Abbildung in CAPPy definiert wurde und wenn nicht, erzeugen sie eine dynamische Abbildungsvorschrift. Dadurch ist eine gültige Abbildung als Rückgabewert sichergestellt, sofern auch die Eingabe gültig ist. Die Methode `createStaticMapperByClass` erzeugt eine statische PYTHON-zu-PROLOG-

Listing 5.10: Statische Abbildungsklasse für `employee/5`

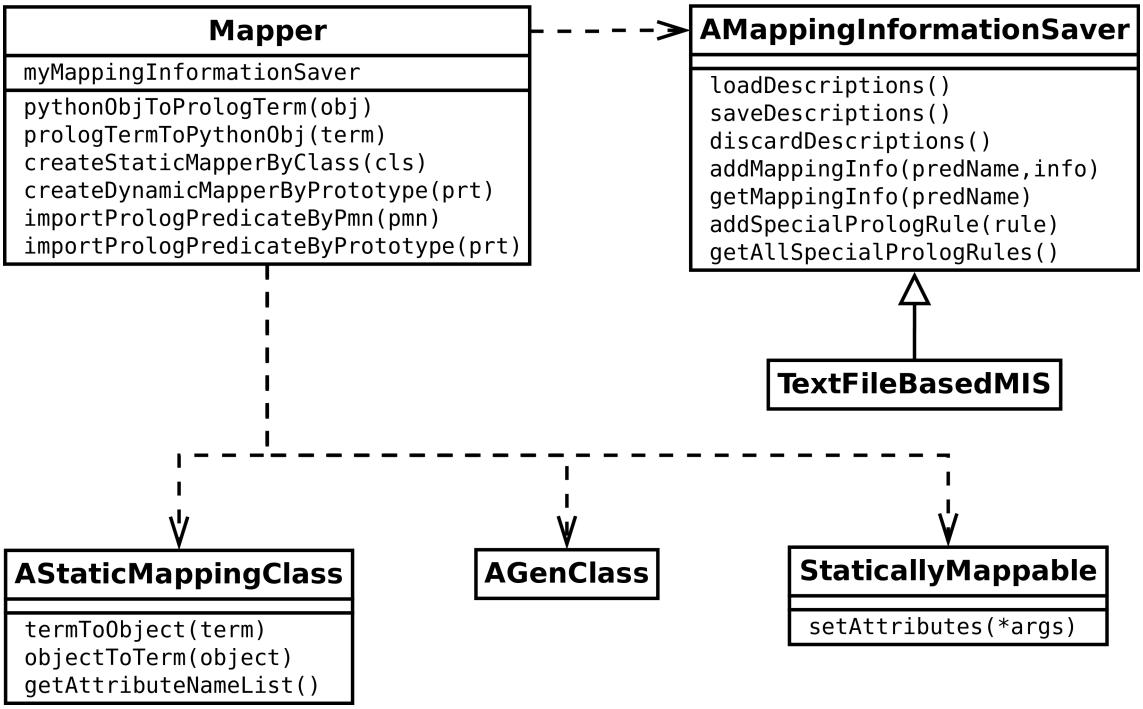
```
from PPMapper.AGenClass import AGenClass
from PPMapper.StaticallyMappable
 ↪ import StaticallyMappable
from AGenClasses.Address import Address

class employee(AGenClass, StaticallyMappable):

 def __init__(self, name : str, valid : int,
  ↪ dep_nr : bool, overtime : float, address : int):
  self.setAttributes(name, valid, dep_nr, overtime,
   ↪ address)

 def setAttributes(self, name : str, valid : int,
  ↪ dep_nr : bool, overtime : float, address : int):
  self.name = name
  self.valid = valid
  self.dep_nr = dep_nr
  self.overtime = overtime
  self.address = address
```

Abbildung und bedarf als Parameter eine PYTHON-Klasse, die eine Unterklasse von `StaticallyMappable` ist. Die erzeugten Abbildungsklassen werden in ein vorgegebenes PYTHON-Pakte geschrieben und automatisch geladen. `createDynamicMapperByPrototype` erzeugt eine dynamische PYTHON-zu-PROLOG-Abbildung. Sie bedarf als Parameter ein PYTHON-Objekt, das ein Prototyp der abzubildenden Klasse ist. Die Abbildungsvorschrift ist dabei eine Festlegung, welches Attribut der PYTHON-Klasse mit welcher Komponente eines PROLOG-Prädikates korrespondiert. Gegebenenfalls werden auch Konvertierungsregeln für die PROLOG-Fakten generiert. Die Methode `importPrologPredicateByPmn` erzeugt eine statische PROLOG-zu-PYTHON-Abbildung. Als Parameter erwartet die Methode eine PMN-Signatur eines PROLOG-Prädikats. Sie muss dabei als vor-verarbeitete PROLOG-Struktur vorliegen, wie sie von PPGATEWAY erstellt wird. PPMAPPING generiert daraus eine PYTHON-Klasse, die das Prädikat in PYTHON repräsentiert. Die erzeugten Klassen sind immer Unterklassen von `AGenClass`, wodurch gekennzeichnet wird, dass die Klassen von CAPPy automatisch generiert wurden. Sie wird in ein vorgegebenes PYTHON-Pakte geschrieben und automatisiert geladen. Anschließend wird die Methode `createStaticMapperByClass` mit der erzeugten Klasse als Parameter aufgerufen, um eine Abbildungsklasse zu erstellen. `importPrologPredicateByPrototype` erzeugt eine dynamische PROLOG-zu-PYTHON-Abbildung. Als Parameter erwartet die Methode ein PROLOG-Prädikat, das als vor-verarbeitete PROLOG-Struktur vorliegen muss, wie sie von PPGATEWAY erstellt wird. Hieraus wird der Name und die Stelligkeit des Prädikats analysiert und eine einfache Klassen-Repräsentation in PYTHON generiert.

Abbildungsverwaltung. Um eine möglichst einfache Benutzbarkeit von PPMAPPING zu gewährleisten, muss zur Abbildung einer Instanz nur eine der Methoden `pythonObjToPrologTerm` und `prologTermToPythonObj` aufgerufen werden. Der Benutzer muss sich dadurch z. B. nicht darum kümmern, ob bereits eine statische Abbildung für die Instanz vorliegt und die entsprechende Abbildungsklasse aufrufen.

Um dies zu ermöglichen enthält PPMAPPING eine Verwaltungseinheit für die Abbildungen. Die Klasse `AMappingInformationSaver` stellt eine Abstraktion dieser Verwaltungseinheit dar. Eine wichtige Eigenschaft von ihr ist, dass nicht nur Abbildungsvorschriften verwaltet werden, die seit Beginn des Programmablaufs erstellt wurden. Sie erfasst vielmehr auch Abbildungsvorschriften über die Terminierung eines CAPPy-Prozesses hinaus. Hierzu ist eine persistente Speicherung dieser Informationen nötig. CAPPy unterstützt zunächst nur eine Implementation hiervon, die die Daten in einer einfache Textdatei verwaltet. Da die restliche Logik von dieser konkreten Speicherform abgetrennt ist, können beliebe weitere Imple-

Abbildung 5.2: Klassenstruktur von PPMapping (vereinfacht)



mentierungen für andere Formate eingefügt werden. Denkbar sind zum Beispiel eine Speicherung in `XML`-Dateien oder in PROLOG-Datenbanken.

Mit den Methoden `loadDescriptions`, `saveDescriptions` und `discardDescriptions` können die persistent gespeicherten Informationen geladen, gespeichert oder verworfen werden. Die Methoden `addMappingInfo` und `addSpecialPrologRule` erlauben eine Abbildungsvorschrift oder eine zusätzliche PROLOG-Regel hinzuzufügen und können dann mit den Methoden `getMappingInfo` und `getAllSpecialPrologRules` abgefragt werden. Als Schlüssel zum Abfragen der Abbildungsvorschriften wird immer der Name des PROLOG-Prädikats genutzt. Sollen die Informationen einer Klasse abgefragt werden, so existiert in PPMAPPING eine Vorschrift, wie aus dem Klassenbezeichner ein eindeutiger Bezeichner in Form eines PROLOG-Prädikats folgt.

5.4 Die Abfragekomponente PPQuery

PPQUERY steht für PROLOG-PYTHON-QUERY und ist die Abfragekomponente von CAPPY. Ihre Aufgabe ist es eine Abfrage in PPQL in eine PROLOG-Abfrage zu überführen. Hierzu wird in diesem Abschnitt gezeigt, wie die Elemente PPQL-Abfrage in äquivalente PROLOG-Elemente übersetzt werden können. Anschließend wird die Implementierung der Übersetzungskomponente vorgestellt.

5.4.1 Ppql-zu-Prolog-Übersetzung

Damit eine Abfrage in PPQL in eine PROLOG-Abfrage übersetzt werden kann, muss definiert werden, wie die einzelnen Strukturen in PPQL auf äquivalente PROLOG-Strukturen übersetzt werden können. Dies wird in diesem Abschnitt durchgeführt.

Die Abfragetypen in PPQL sind immer Objekte. Deswegen benutzt PPQUERY die PPMAPPING-Komponente, um für sie eine PROLOG-Repräsentation zu bestimmen.

Die Vergleichsoperatoren `<`, `>`, `>=`, `<=` und `==` in PPQL sind syntaktisch identisch mit den PROLOG-Äquivalenten. Lediglich der Vergleichsoperator `!=` in PPQL wird auf `\==` in PROLOG abgebildet. Die erlaubten hartcodierter Werte in PPQL können wie im Abschnitt 5.3.1 dargestellt in eine PROLOG-Struktur überführt werden. Die logischen Verknüpfungen `and`, `or` und `not` in PPQL können zu den PROLOG-Strukturen "`,`", "`;`" und "`\+`" übersetzt werden. Die Klammern in einer PPQL-Abfrage können unverändert übernommen werden.

Listing 5.11: Prolog-Regeln für den `in`-Operator

```
cappy_ppql_inop(Val, L) :- is_list(L), member(Val, L)
cappy_ppql_inop(Val, tupel(L)) :- member(Val, L)
cappy_ppql_inop(Val, set(L)) :- member(Val, L)
cappy_ppql_inop(Val, frozenset(L)) :- member(Val, L)
```

Listing 5.12: Prolog-Regeln für die `dict`-Struktur

```
cappy_ppql_dict(dict(Keys, Elements), Key, Elem) :-
 ↪ cappy_ppql_indexList(Key, Keys, Pos),
 ↪ cappy_ppql_indexList(Elem, Elements, Pos)
cappy_ppql_indexList(Elem, [Elem|_], 0).
cappy_ppql_indexList(Elem, [_|T], Idx) :-
 ↪ cappy_ppql_indexList(T, Elem, IdxDecr),
 ↪ Idx is IdxDecr + 1.
```

Wie Container-Typen aus Ppql in Prolog-Strukturen übersetzt werden können, kann dem Abschnitt 5.3.1 entnommen werden. Damit der `in`-Operator aus Ppql nach Prolog übersetzt werden kann, lädt CAPPy, sobald eine Verbindung mit einem Prolog-Interpreter etabliert wurde, in eine spezielle Prolog-Bibliothek ein, siehe Listing 5.11. Diese enthält einfache Prolog-Regeln, mit denen der `in`-Operator in Prolog ausgedrückt werden kann.

Auch für das assoziative Datenfeld `dict` in Ppql enthält die Prolog-Bibliothek eine Regel, mit der ein Eintrag aus ihm abgefragt werden kann, siehe Listing 5.12. Sie besagt, dass der Schlüssel und das Element im Datenfeld an der selben Stelle in der jeweiligen Listen der Prolog-Repräsentation stehen.

Eine `omit`-Struktur wird nicht übersetzt. Vielmehr wird das Attribut, das mittels `omit` deklariert wird, in der Abfrage lediglich durch die anonyme Variable `"_"` repräsentiert.

5.4.2 Implementierung der Übersetzungskomponente

In diesem Abschnitt wird die Implementierung der Übersetzungskomponente eingeführt. Sie wurde mithilfe reflexiver Funktionen in Python und dem Parsergenerator Antlr implementiert.

Listing 5.13: Company-Datenbank in Prolog-Notation

```
predicate(customer, [customer_id, name, address],
 ↪ [int, str, address])
predicate(address, [post_code, city], [str, str])
customer(1001, 'Mueller', address('50667', 'Koeln')).
customer(1002, 'Beacker', address('80331', 'Muenchen')).
predicate(item, [item_id, description], [int, str])
item(1, 'Drucker').
item(2, 'Bildschirm').
predicate(order, [order_id, item_list, customer_id],
 ↪ [int, list(int), int])
order(1, [2], 1001).
```

Listing 5.14: Abfrage an die Company-Datenbank in Ppql

```
def customer_query(c : Predicate_customer,
 ↪ i : Predicate_item, o : Predicate_order):
 c.customer_id == o.customer_id
  ↪ and (i.item_id in o.item_list)
  ↪ and (i.description == 'Drucker'
  ↪ or i.description == 'Bildschirm')
  ↪ and omit(c.address)
```

Die Grundlage für die Antlr-Grammatik ist eine gewöhnliche Python-Grammatik, weil Ppql eine interne DSL von Python ist. Es wurden lediglich überflüssige Regeln entfernt und kleinere Regeln hinzugefügt, mit denen das Parsen einer Ppql-Abfrage erleichtert wird.

Im Folgenden werden die einzelnen Schritte aufgeführt, mit denen PPQuery eine Ppql-Abfrage in eine Prolog-Abfrage übersetzt. Um die Schritte zu veranschaulichen, werden sie an einer Abfrage auf der Beispieldatenbank *Company* demonstriert. Die Datenbank kann in Listing 5.13 eingesehen werden und beinhaltet Pmn-Notationen zu allen Prolog-Prädikaten. Das Ziel der Beispielabfrage ist es, alle Kunden zu bestimmen, die einen Bildschirm oder einen Drucker gekauft haben, wobei ausgegangen wird, dass die Datenbankstruktur mittels statischer Prolog-zu-Python-Abbildung in Python-Klassen abgebildet wurde. Eine Ppql-Repräsentation dieser Abfrage findet sich in Listing 5.14.

Die einzelnen Schritte der Ppql-zu-Prolog-Übersetzung lauten:

Listing 5.15: Initialisierung einer PPQL-zu-PROLOG-Übersetzung

```
omitList = []
substitionsVariablesNo = 0
substitionsStr = ''
stack = []
```

Listing 5.16: Setzten der `dict`-Struktur `queryVariables`

```
queryVariables = {
 ↪ 'c': {'customer_id' : QV0_0, 'name' : QV0_1,
 ↪ 'address': QV0_2},
 ↪ 'i': {'item_id' : QV1_0, 'description' : QV1_1},
 ↪ 'o': {'order_id' : QV2_0, 'item_list' : QV2_1,
 ↪ 'customer_id' : QV2_2}}
```

1. **Schritt** Die Variablen für die Übersetzung werden initialisiert, siehe Listing 5.15

2. **Schritt** Mittels Reflexion wird aus der PPQL-Abfrage die Funktionssignatur abgefragt. Aus ihr werden die Abfragetypen extrahiert. Mit diesen Informationen wird die PPMAPPING-Komponente angesprochen und eine Attributliste angefordert. Daraus wird eine verschachtelte `dict`-Struktur `queryVariables` generiert, die angibt, wie die einzelnen Attributsbezeichner der Abfragetypen zu PROLOG-Variablen übersetzt werden sollen. Der erste Schlüssel ist dabei der Abfragetyp und der zweite Schlüssel der Attributsbezeichner. Die PROLOG-Variablen haben hierbei die folgende Form: Die Zeichen `QV`, gefolgt von der internen Nummerierung des Abfragetyps, gefolgt von einem Unterstrich, gefolgt von der Position im entsprechendem Prädikat. Diese Form der Variablennamen hat den Hintergrund, dass die einzelnen Variablen nach der Auswertung so eindeutig und einfach zugewiesen werden können. Die resultierende Struktur aus der Beispielabfrage kann in Listing 5.16 eingesehen werden.

3. **Schritt** Mittels reflexiver Methoden wird der Funktionsrumpf eingelesen. Um den Hauptteil der Abfrage nach PROLOG zu übersetzten, wird aus dem Funktionsrumpf und einem PPQL-Parser ein Syntaxbaum erstellt. Der *Walker* läuft diesen ab, wobei immer, wenn er eine Regel betritt oder verlässt, ein *Listener* aufgerufen wird. Ob er beim Verlassen oder Betreten aufgerufen wird, hängt von der Regel ab. Jede Regel beschreibt eine syntaktische Einheit

Listing 5.17: Interner Zustand nach Schritt 3

```
omitList = [QV0_1]
substitionsVariablesNo = 0
substitionsStr = ''
stack = ['QV0_0', '==', 'QV2_2', ',',
 ↪  '(', 'cappy_ppql_inop(QV1_0, QV2_1)', ')', ',',
 ↪ '(', 'QV1_1', '==', '\'Drucker\'', ';',
 ↪ 'QV1_1', '==', '\'Bildschirm\'', ')', ',', 'true']
```

Listing 5.18: Stack als Zeichenkette

```
stackString = 'QV0_0 == QV2_2,
 ↪ (cappy_ppql_inop(QV1_0, QV2_1)),
 ↪ (QV1_1 == \'Drucker\' ; QV1_1 == \'Bildschirm\'),
 ↪ true
```

von PPQL. Wie in Abschnitt 5.4.1 beschrieben, wird für jede dieser Einheiten eine Übersetzung erstellt. Diese wird auf einen Stack gelegt. Der Grund dafür ist, dass die Syntax für die PPQL-Strukturen `in` und `dict` eine Infix- bzw. Pre- und Postfix-Notation besitzen, während die PROLOG-Struktur eine reine Präfix-Notation besitzt. Wenn sie übersetzt werden, dann nehmen sie die entsprechenden Elemente vom Stack und legen ein neues Element auf ihn, das die komplette Struktur abbildet. Die `dict`-Strukturen werden dabei zu Substitutionsvariablen übersetzt. Diese werden nach der Form `SV_` gefolgt von einer Id, die in der `substitionsVariablesNo` abgefragt wird, gebildet. Anschließend wird `substitionsVariablesNo` inkrementiert und die zu substituierende Übersetzung an `substitionsStr` angehängt. Wird ein Attributsbezeichner mittels `omit` deklariert, so wird die PROLOG-Übersetzung zu der Liste `omitList` hinzugefügt und der Ausdruck selbst mit `true` übersetzt. Nachdem der *Walker* den Baum vollständig abgelaufen ist, existiert auf dem Stack eine teilweise Übersetzung der Abfrage. Wird die Beispielabfrage verarbeitet, so sieht der interne Zustand nach diesem Schritt von PPQUERY wie in Listing 5.17 aus.

4. Schritt Nach dem der Hauptteil vor-verarbeitet wurde kann durch eine Nachverarbeitung die Abfrage generiert werden. Die Elemente in der Stack-Variablen, werden zu einer einheitlichen Zeichenkette umgeformt, wie in Listing 5.18. Für jeden Abfragetyp wird der Abfrage noch das korrespondierende PROLOG-Prädikat hinzugefügt und die Abfrage-Variablen gemäß

Listing 5.19: Setzten der `dict`-Struktur `queryVariables`

```
queryVariablesString = customer(QV0_0, QV0_1, _),
 ↪ item(QV1_0, QV1_1), order(QV2_0, QV2_1, QV2_2)
```

Listing 5.20: Übersetzte PPQL-Anfrage

```
customer(QV0_0, QV0_1, _), item(QV1_0, QV1_1),
 ↪ order(QV2_0, QV2_1, QV2_2), QV0_0 == QV2_2,
 ↪ (cappy_ppql_inop(QV1_0, QV2_1)),
 ↪ (QV1_1 == 'Drucker' ; QV1_1 == 'Bildschirm'), true
```

der `queryVariables`-Struktur gesetzt. Ist eine Abfrage-Variable in der Liste `omitList` enthalten, wird es durch die anonyme Variable `"_"` ersetzt. Aus der Beispielabfrage ergibt sich damit eine Zeichenkette, wie in Listing 5.19. Durch die Konkatenation der Zeichenketten ergibt sich die endgültige PROLOG-Abfrage, wie in Listing 5.20.

5. Schritt Nachdem die PPQL-Abfrage nach PROLOG übersetzt wurde, kann sie an das Gateway weitergeleitet werden. Je nachdem was das Ziel der Abfrage war, wird zurückgemeldet, dass eine solche Instanz existiert oder eine Instanz mit den zurückgebenden Zuweisungen zu den Variablen erzeugt. Im letzten Fall würde aus der Beispielabfrage ein Ergebnis, wie in Listing 5.21 zurückgeliefert werden

Listing 5.21: Ergebnis der PPQL-Anfrage in PYTHON-Form

```
result = [(Predicate_customer(1001, 'Mueller', None),
 ↪ Predicate_item(2, 'Bildschirm'),
 ↪ Predicate_order(1, [2], 1001))]
```

Anpassungen für Rdf-Abfragen. Damit einfachere RDF-Abfragen möglich sind, sind lediglich an zwei Stellen der Übersetzungskomponente Änderungen vorgenommen worden:

- Wenn in Schritt 2 die Funktionssignatur abgefragt wird, werden sämtliche Typhinweise ignoriert. Es wird stets davon ausgegangen, dass alle Abfragetypen RDF-Objekte sind. Diese sind Instanzen zu einer fest in CAPPy eingebauten Klasse.
- Der *Listener* für die ANTLR-Regeln, der die Funktionsaufrufe erkennt, wurde verändert. Ist der "Funktionsaufruf" nicht `omit`, so wird dies als ein RDF-Präfix verstanden. Eine PPQL-Einheit `foaf(Person)` wird zu `foaf:Person` übersetzt und auf den Stack gelegt

6 Diskussion und Fazit

In diesem Kapitel werden die Ergebnisse dieser Arbeit diskutiert und ein Fazit gezogen. Dazu wird zunächst gezeigt, wie die Implementierung von CAPPy getestet wurde. Weiterhin wird ein Ausblick auf zukünftige Entwicklungsmöglichkeiten von CAPPy gegeben. Anschließend wird gezeigt, dass die Ziele der Arbeit erfüllt wurden. Zum Schluss wird das Ergebnis dieser Arbeit zusammengefasst.

6.1 Softwaretest

In diesem Abschnitt wird der Softwaretest von CAPPy betrachtet. CAPPy wurde mittels Unittests auf Korrektheit überprüft. Hierfür wurde für jede der drei Hauptkomponenten PPGateway, PPMapping und PPQuery jeweils eine Testklasse erstellt, die die bereits eingeführte Implementierung mit verschiedenen Testfälle überprüft. Aufgrund der Komplexität einiger generierter Strukturen, müssen manche Testergebnisse manuell untersucht werden.

PPGateway Es wurden für alle Methoden, die in der Klasse Ppg definiert sind, Testfälle erstellt. Für die Methode `call` wurden viele Testfälle mit unterschiedlichen Fallkonstruktionen erstellt. Es existieren Abfragen, die zu wahr oder falsch ausgewertet können. Weiterhin finden sich einige Abfragen, die Variablen enthalten und deren Ergebnis eine Zuweisung zu diesen ist.

PPMapping Sowohl für die statische, als auch für die dynamische Abbildung von Prolog nach Python und umgekehrt wurden Testfälle erstellt. Hierbei wurden auch aufeinander aufbauen Strukturen getestet. Dass die Abbildungsinformationen vollständig gespeichert und geladen werden können, wurde ebenfalls abgedeckt.

PPQuery Es wurden verschiedene Abfragen in Ppql erstellt. Hierbei gibt es einfache Abfragen, als auch komplexe mit vielen Operatoren und Verschaltungen. Auch die `omit`-Operatoren wurden getestet. Um die Rdf-Funktionalität der Ppql-Abfragen zu testen, wurde verschiedene Abfragen auf einer handlichen Rdf-Datenbank durchgeführt.

Die einzelnen Testfälle können im Programmcode von CAPPy eingesehen werden. Alle Testfälle können mit der bestehenden Implementierung erfolgreich durchgeführt werden, was ein Indiz für die Korrektheit der Implementierung ist.

6.2 Weitere Entwicklungsmöglichkeiten von CAPPy

Auch wenn mit CAPPy bereits ein solides Framework für die Integration von Prolog in Python vorliegt, gibt es noch Verbesserungsmöglichkeiten und neue Features, die hinzugefügt werden können. In diesem Abschnitt werden einige interessante Möglichkeiten hierfür aufgezeigt.

1. Es können weitere Gateways zu anderen Prolog-Interpretern hinzugefügt werden, wie zum Beispiel Gnu Prolog. Dazu muss einfach eine Unterklasse von Ppg erstellt werden, die die entsprechenden Funktionen für den neuen Interpreter implementiert. Die restliche Anwendungslogik von CAPPy ist sauber den einzelnen Interpretern getrennt, weswegen keine weiteren Anpassungsschritte vorgenommen werden müssen.

2. Das SwiPrologGateway ist bisher auf Abfragen ausgelegt, die nicht auf die Standartausgabe schreiben. Soll eine Prolog-Anwendung eingebunden werden, die dies benutzt, so kommt es zu Fehlern im SwiPrologGateway, da es keine solche Ausgaben erwartet. Eine Verbesserungsmöglichkeit wäre es, wenn ein anderer Ansatz zur Interaktion mit einem Prolog-Interpreter gefunden wird, als die textbasierte Verarbeitung des Ausgabestreams.

3. PPMapping nutzt bisher nur eine einfache Testdatei, um die Abbildungsinformationen persistent zu speichern. Es bietet sich die Möglichkeit an, die Daten in einer strukturierten Form, wie in einer Xml-Datei, zu speichern oder die Informationen zu der Prolog-Datenbank hinzufügen. Die letztere Möglichkeit hat den Vorteil, dass alle persistenten Daten zentral mit der angebundenen Prolog-Datenbank gespeichert werden würden. Dies wäre bei einer gemeinschaftlichen Entwicklung von Vorteil.

4. Die Abfragesprache Ppql kann um neuer Funktionalitäten erweitert werden. Ppql könnte auch um weitere Abfragemechanismen erweitert werden. Eine Möglichkeit wäre es, bestimmte Python-Strukturen zu erlauben. Als Beispiel sei hier angenommen, man möchte die Abfrage `studentQuery1` aus Listing 4.4 so abändern, dass nicht mehr der Nachname "Mustermann" ein Kriterium ist, sondern, dass der Nachname den Teilstring "mann" enthält.

Listing 6.1: PPQL-Abfrage mit Teilstring-Bedingung

```
1 def studentQuery1_substr (s : Student):
2   "mann" in s.last_name and s.age >= 18
```

Eine solchen Teilstring-Test führt man in PYTHON mittels dem Operanden `in` aus. Die abgeänderte Abfrage sähe dann wie in 6.1 aus. Nicht alle PYTHON-Strukturen lassen sich unmittelbar in ISO-PROLOG-kompatible Konstrukte umwandeln. Deswegen muss die Übersetzungskomponente bei solchen Erweiterungen dies berücksichtigen. Dazu wird jede Bedingung vorab geprüft, ob sie entsprechend abgebildet werden kann. Ist dem nicht der Fall, so wird sie zunächst ignoriert, so dass es auch Rückgabewerte gibt, die die Ursprungsbedingungen nicht erfüllen. Die Übersetzungskomponente wird in diesem Fall erneut aufgerufen und sortiert alle Ergebnisse aus, die nicht mit der Abfrage konform sind. Perspektivisch wäre eine möglichst weitgehende Unterstützung der PYTHON-Syntax wünschenswert, weil der Benutzer so mehr Freiheiten beim Erstellen der Abfrage hat und so noch besser bei seinen bekannten Konzepten abgeholt wird.

5. Die Abfragetypen in PPQL sind bisher nur Objekte. Eine neuer Ansatz wäre es, wenn nicht nur Objekte, sondern auch einzelne Werte als Abfragetyp definiert werden können. So ist es z. B. in SQL üblich, dass einzelne Werte und keine kompletten Tupel ausgegeben werden können. Hierzu muss evaluiert werden, inwiefern dadurch eine einfache Benutzbarkeit noch gewährleiste ist. Im positiven Fall müsste dann eine neue Definition der Abfragesprache PPQL gefunden werden und PPQUERY-Komponente umgeschrieben werden.

6. In CAPPY werden bisher die Abfragen zur Laufzeit übersetzt. Wenn bereits zur Kompilierzeit feststeht, wie die Abfrage lautet, so könnte sie bereits zu diesem Zeitpunkt übersetzt werden. Dadurch würde sich die Verarbeitungsgeschwindigkeit von CAPPY erhöhen. Die bisherige Möglichkeit zur dynamischen Übersetzung sollte aber in jeden Fall erhalten bleiben. Denn dass Abfragen zur Laufzeit generiert werden können, entspricht der PYTHON-typischen Dynamik.

6.3 Validierung

In diesem Abschnitt wird erklärt, dass mit CAPPy ein neues Intergrationsframework für Prolog in Python erschaffen wurde, das eine bessere Integration als mit den bisherigen Werkzeugen ermöglicht.

CAPPy ist, im Gegensatz zu den meisten Werkzeugen, mit den aktuellen Versionen von Swi-Prolog und Python kompatibel, was ein Vorteil von CAPPy darstellt. Insbesondere für neu entwickelte Software ist es für eine gute Wartbarkeit entscheidend, dass die Software in einer modernen Umgebung ausgeführt werden kann. Prinzipiell ist CAPPy mit beliebigen Prolog-Interpretern kompatibel. Derzeit implementiert es jedoch nur eine Unterstützung für Swi-Prolog. Dennoch kann eine Unterstützung zu neuen Interpretern leicht hinzugefügt werden. Da die restliche Programmlogik von den konkreten Interpretern sauber getrennt ist, ergibt sich der Vorteil, dass mit CAPPy unabhängig von einem konkreten Prolog-Interpreter entwickelt werden kann. Wird zu einem späteren Entwicklungszeitpunkt entschieden einen anderen Interpreter zu benutzen, so hat das auf den erstellten Code keinen Einfluss.

Indem CAPPy korrespondierende Strukturen in Prolog und Python etablieren kann, ergibt sich für den Python-Benutzer ein besseres Verständnis einer Prolog-Datenbank. Die Python-Strukturen bestehen dabei nicht nur aus nativen Konstrukten, sie werden auch in typischer Form eingesetzt. Es ergeben sich verschiedene Einsatzmöglichkeiten durch die Abbildungskomponente:

- Ein Entwickler einer Prolog-Datenbank annotiert die Datenbank mit der Pmn. Das ist eine interne DSL für Prolog, die das Ziel hat, eine einfache Abbildung einer Prolog-Datenbank in eine typische Höhere Programmiersprache zu erleichtern. Der Benutzer kann die statische Prolog-zu-Python-Abbildung nutzen, um eine Repräsentation der Datenbank in Python zu ermöglichen. Fortgeschrittene Benutzer können in diesem Fall auch den Abbildungsmechanismus nach Belieben anpassen.
- Liegt keine annotierte Datenbank vor, so kann der Benutzer die Datenbank mit einem grafischen Tool durchforsten, wie z. B. mit Declare. Hierdurch erhält er Kenntnisse über die Datenbankstruktur und kann die dynamische Prolog-zu-Python-Abbildung nutzen, um eine Repräsentation der Datenbank in Python zu ermöglichen.
- Der Benutzer kann auch in Python eine Datenbank erstellen und sie mittels CAPPy nach Prolog abbilden lassen. Hierdurch kann er die restlichen

Eigenschaften von CAPPy nutzen und dadurch eine leicht zu bedienende Datenbankkomponente in Python erhalten.

Mit den bisherigen Werkzeugen ist eine Bedienung oft nicht intuitiv möglich. In der Regel bauen sie ein Prolog-Interface in Python nach. Der Benutzer muss deshalb über Prolog-Kenntnisse verfügen, um sinnvoll mit ihnen arbeiten zu können. Mit CAPPy kann eine Prolog-Datenbank auf intuitive Weise abgefragt werden. Hierfür implementiert es die Ppql, einer objektorientierten Abfragesprache, die stark an die Python-Syntax angepasst ist. Nachdem der Benutzer anhand der korrespondierenden Klassenstruktur in Python eingesehen hat, kann er eine Abfrage erzeugen, in dem er Bedingungen stellt, die die Objekte dieser Klassen erfüllen sollen. Die Ppql-Abfrage wird automatisiert in eine Prolog-Abfrage übersetzt und als Ergebnis werden Objekte zurückgeliefert, die diese Eigenschaft erfüllen. CAPPy hat daher eine geringe Einstiegshürde für Python-Programmierer. Auf diese Weise wird ein Zugang zu Prolog für Programmierer eröffnet, die ansonsten auf Prolog verzichten würden und stattdessen auf eine weiter-verbreitete Lösungen setzen würden.

Durch eine besondere Unterstützung der Rdf-Bibliothek ClioPatria, können mit CAPPy auch elegant Rdf-Daten anhand der Ppql verarbeitet werden. Durch die Testfälle wurde gezeigt, dass die Ontologie Yago damit einfach in Python eingebunden werden kann. Die Daten können bisher nur durch Ppql-Abfragen durchforstet werden. Soll ein tiefergehendes Verständnis der Rdf-Tripel erreicht werden, so muss auf andere Werkzeuge zurückgegriffen werden, wie z. B. Visualisierungswerkzeuge.

Anhand dieser Eigenschaften lässt sich konstatieren, dass mit der Implementierung von CAPPy die Ziele dieser Arbeit erfüllt wurden.

6.4 Fazit

Mit CAPPy wurde ein neuer und einheitlicher Ansatz vorgestellt, mit dem eine intuitive Integration von Prolog in Python ermöglicht wird. Hierzu wurde gezeigt, dass es viele interessante Prolog-Anwendungen gibt, die einer breiteren Nutzerschaft zur Verfügung gestellt werden sollten. In Fallstudien wurde gezeigt, dass die bisherigen Werkzeuge dafür aber nicht ausreichten. Deswegen wurde die Machbarkeit des Frameworks CAPPy nach dem Vorbild von CAPJa untersucht, ein Designvorschlag erstellt und evaluiert. Durch die Testfälle konnte bestätigt werden, dass die Implementierung von CAPPy korrekt ist. Die korrespondierenden Strukturen in Prolog und Python ermöglichen einen einfachen Zugang zu der

Datenbankstruktur, und mit PPQL wurde eine leicht zugängliche Abfragesprache eingeführt. Durch eine besondere Unterstützung von RDF-Abfragen, können auch RDF-Ontologien, wie YAGO, leicht in PYTHON integriert werden. CAPPY kann zukünftig um weitere Komponenten erweitert werden, wodurch noch mehr Funktionalitäten von PROLOG intuitiv in PYTHON nutzbar gemacht werden können. CAPPY ist damit ein guter Einstieg in die Programmiersprache PROLOG und soll dabei helfen, das Interesse an dieser Programmiersprache zu fördern.

Literaturverzeichnis

[CB14] Curé, Olivier ; Blin, Guillaume: Rdf *Database Systems: Triples Storage and* Sparql *Query Processing.* Elsevier Science, 2014

[Cho56] Chomsky, N.: Three models for the description of language. In: *IRE Transactions on Information Theory* 2 (1956), Nr. 3, S. 113–124

[Con] `https://www.w3.org/2001/sw/`

[Fow10] Fowler, Martin: *Domain-specific languages.* Pearson Education, 2010

[GM10] Gabbrielli, Maurizio ; Martini, Simone: *Programming Languages: Principles and Paradigms.* Springer Science & Business Media, 2010

[HSBW13] Hoffart, Johannes ; Suchanek, Fabian ; Berberich, Klaus ; Weikum, Gerhard: Yago2: A spatially and temporally enhanced knowledge base from Wikipedia. In: *Artificial Intelligence* 194 (2013), Nr. Supplement C, S. 28 – 61

[IBNW09] Ireland, Christopher ; Bowers, David ; Newton, Michael ; Waugh, Kevin: A classification of object-relational impedance mismatch. (2009), S. 36–43

[Inf] `https://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/yago/`

[ISOa] `https://www.iso.org/standard/21413.html`

[ISOb] `https://www.w3.org/TR/rdf-sparql-query/#WritingSimpleQueries`

[MBS14] Mahdisoltani, Farzaneh ; Biega, Joanna ; Suchanek, Fabian: Yago3: A knowledge base from multilingual Wikipedias. In: *7th Biennial Conference on Innovative Data Systems Research (CIDR Conference)*, 2014

[MJ99] Martin, James ; Jurafsky, Daniel: *Speech and language processing.* Prentice-Hall, 1999

[MJD96] Malenfant, Jacques ; Jacques, Marco ; Demers, Franois N.: A tutorial on behavioral reflection and its implementation. In: *Proceedings of the Reflection Conference* Bd. 96, 1996, S. 1–20

[Ost17] Ostermayer, Ludwig: *Integration of* Prolog *and* Java *with the Connector Architecture* CAPJa, Diss., 2017

[Par13] Parr, Terence: *The definitive* Antlr *4 reference*. Pragmatic Bookshelf, 2013

[Pau11] Paulheim, Heiko: *Ontology-based Application Integration*. Springer New York, 2011

[pex] `https://pexpect.readthedocs.io/en/stable/`

[PS02] Pereira, Fernando ; Shieber, Stuart: Prolog *and natural-language analysis*. Microtome Publishing, 2002

[pys] `https://github.com/tjvr/pyswip`

[Sei15] Seipel, Dietmar: *Deduktive Datenbanken und Logikprogrammierung*. Skriptum. Universität Würzburg, 2015

[Sei17] Seipel, Dietmar: *The Declare Developers' Toolkit (*Ddk*)*. 31.08.2017 `http://www1.informatik.uni-wuerzburg.de/database/DisLog/`

[SHM08] Schneider, Gerold ; Hess, Michael ; Merlo, Paola: *Hybrid long-distance functional dependency parsing*. Unpublished Diss., 2008

[SKW07] Suchanek, Fabian ; Kasneci, Gjergji ; Weikum, Gerhard: Yago: A core of semantic knowledge unifying Wordnet and Wikipedia. In: *International World Wide Web Conference*, 2007, S. 697–706

[SKW08] Suchanek, Fabian ; Kasneci, Gjergji ; Weikum, Gerhard: Yago: A Large Ontology from Wikipedia and Wordnet. In: *Web Semantics: Science, Services and Agents on the World Wide Web* 6 (2008), Nr. 3, S. 203–217

[SNA17] Seipel, Dietmar ; Nogatz, Falco ; Abreu, Salvador: Domain-specific languages in Prolog for declarative expert knowledge in rules and ontologies. In: *Computer Languages, Systems & Structures (Comlan)* (2017). `https://doi.org/10.1016/j.cl.2017.06.006.`. – Elsevier, 2017

[SSVW09] Sennrich, Rico ; Schneider, Gerold ; Volk, Martin ; Warin, Martin: A new hybrid dependency parser for German. In: *Proceedings*

of the German Society for Computational Linguistics and Language Technology 115 (2009), S. 124ff

[SVS13] Sennrich, Rico ; Volk, Martin ; Schneider, Gerold: Exploiting Synergies Between Open Resources for German Dependency Parsing, POS-tagging, and Morphological Analysis. (2013), S. 601–609

[swi] `http://www.swi-prolog.org/contrib`

[Tar00] Tarvainen, Kalevi: *Einführung in die Dependenzgrammatik*. Bd. 35. Walter de Gruyter, 2000

[tioa] `https://www.tiobe.com/tiobe-index/programming-languages-definition`

[tiob] `https://www.tiobe.com/tiobe-index`

[Ull] Ullenboom, Christian: Java *ist auch eine Insel*. `http://openbook.rheinwerk-verlag.de/javainsel`

[vPa] van Rossum, Guido ; Python Development Team: Python *Frequently Asked Questions*. `https://docs.python.org/3/archives/python-3.6.3rc1-docs-pdf-a4.zip`

[vPb] van Rossum, Guido ; Python Development Team: *The* Python *Language Reference*. `https://docs.python.org/3/archives/python-3.6.3rc1-docs-pdf-a4.zip`

[vPc] van Rossum, Guido ; Python Development Team: Python Tutorial. `https://docs.python.org/3/archives/python-3.6.3rc1-docs-pdf-a4.zip`

[WBHv16] Wielemaker, Jan ; Beek, Wouter ; Hildebrand, Michiel ; van Ossenbruggen, Jacco: ClioPatria: A Swi-Prolog Infrastructure for the Semantic Web. In: *Semantic Web* 7 (2016), Nr. 5, S. 529–541

[Wik] `https://en.wikipedia.org/wiki/Special:Statistics`

Index