Implementierung eines Prolog-Adapters für Java

Praktikum von

Johannes Werner

Institut für Software Design und Qualität
Betreuender Mitarbeiter: M.Sc. Stephan Seifermann

1 Einführung

Programmiersprachen haben unterschiedliche Stärken und Schwächen. Das liegt daran, dass Sprachen meist für ein bestimmtes Problem oder eine bestimmte Domäne an Problemen entwickelt werden. Schaut man sich Java und Prolog an, wird dies besonders deutlich. Java ist eine objekt-orientierte Programmiersprache mit dem Ziel, möglichst zugänglich und gleichzeitig möglichst mächtig zu sein. Durch den imperativen Programmierstil ist diese Sprache außerdem für viele Menschen gut verständlich und kann für eine Vielzahl an Problemen eingesetzt werden. Prolog hingegen verfolgt einen ganz anderen Ansatz. Anstatt eine Sequenz an Befehlen zu programmieren, die angeben, *wie* ein Problem gelöst wird, definert man in Prolog lediglich *was* das Problem ist und der Prolog-Interpreter löst es dann. Das ist vorallem dann von Vorteil, wenn man vorrangig logische Probleme lösen will.

Um die Vorteile einer Sprache auch sprachübergreifend nutzen zu können, werden häufig Bibliotheken bereitgestellt, die als Schnittstelle zu anderen Sprachen dienen. Beispielsweise stellen einige Prolog-Interpreter Java-Bibliotheken bereit, die einen einfachen Zugang zu Prolog in Java-Code ermöglichen. Allerdings haben diese Bibliotheken keine einheitliche API, was es erschwert, verschiedene Implementierungen derselben Sprach in einem Programm zu nutzen. Dabei ist eine Auswahl verschiedener Implentierungen genau das, was in manchen Fällen erforderlich sein kann. Gründe dafür können sich per Einsatzszenario ändernde Anforderungen sein, wie beispielsweise Lizenzbedingungen oder Performanceunterschiede. Um dennoch mehrere verschiedene Implemntierungen verwenden zu können, kapselt man die unterschiedlichen APIs in einem Adapter und stellt nach außen hin eine einheitliche API zur Verfügung. Abbildung 1 zeigt, wie ein solcher Adapter shematisch aussieht. Schnittstellen zu einer Zielsprache werden entweder als eine native Bibliothek zur Verfügung gestellt, oder die Zielsprach ist direkt in

der Quellsprache implementiert. Der Adapter sellbst ist in der Quellsprache implementiert und kapselt die unterschiedlichen Schnittstellen-APIs. Dadurch kann das Hauptptogramm einheitlich auf die verschiedenen Implementierungen der Zielsprache zugreifen.

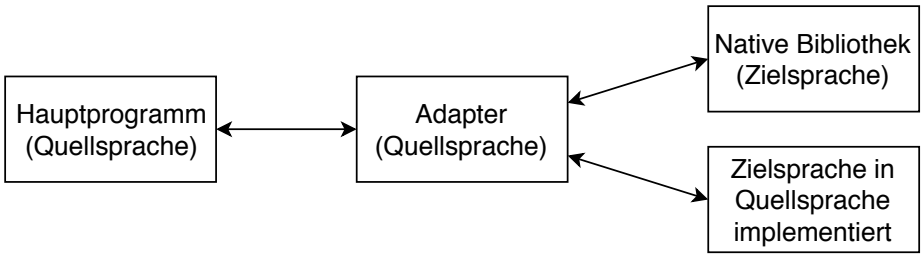

Abbildung 1: Ein Adapter ist in der Quellsprache implementiert und greift entweder auf eine native Bibliothek oder auf eine Quellsprach-Implementierung der Zielsprache zu.

1.1 Projektrahmen

Dieses Praktikum findet im Rahmen des Projektes Trust 4.0[7] statt. Trust 4.0 hat es sich als Ziel gesetzt, Datenschutz in einem Softwaresystem garantieren zu können. Dazu wird mittels eines Plugins in Eclipse ein Architekturmodell des zu prüfenden Softwaresystems erstellt und um Datenflussannotationen ergänzt. Diese Annotation beschreiben, welchen Weg personenbezogne Daten über verschiedene Software-Module hinweg nehmen und aufgrunddessen kann man analysieren, ob es an einer Stelle zu Datenlecks kommen kann. Das Eclipse-Plugin selbst ist in Java implementiert, jedoch hat man sich bei der Datenflussanalyse für Prolog entschieden. Wie bereist weiter oben erläutert, ist Prolog speziell für rein logischen Probleme entwickelt worden, weshalb es sich für die Analyse besser eignet als Java. Um es in Zukunft dem Nutzer zu ermöglichen, von dem Eclipse-Plugin aus die Analyse auf einem frei gewählten Prolog-Interpreter auszuführen, ist es das Ziel dieses Praktikums, einen Prolog-Adapter für Java zu entwickeln.

2 Anforderungen

Wie bereits in Kapitel 1.1 geschildert, soll der Adapater im Projekt Trust 4.0 eingesetzt werden. Daraus egeben sich einige Anforderungen, damit eine Integration in die bestehende Codebasis möglich ist.
Der Adapter soll in dem projekteignen Eclipse-Plugin eingesetzt werden. Dieses Plugin ist unter der Eclipse Plugin Licence (EPL)[2] lizenziert, was zu der Anfordung führt, dass Software, die in dem Plugin zum Einstaz kommt, auch unter EPL oder zu EPL kompatibel lizenziert sein muss. Diese Anforderung gilt ebenso für den Adapter und die durch ihn angebundene Interpreter.
SWI-Prolog[8] ist der quasi Standard-Interpreter in der Prologwelt. Kein anderer Interpreter ist in dem Maße verbreitet und wird auch dementsprchend aktiv weiterentwickelt, weshalb eine Anbindung daran unerlässlich ist. Außerdem muss mindestens ein Weiterer

der angebundenen Interpreter in Java implementiert sein. Der Grund dafür ist, dass das Plugin bei der Auslieferung möglichst wenige Abhängigkeiten auf bereits vorinstallierte Software haben soll. Ein in Java implementierter Interpreter kann in das Plugin-Packet miteingebunden werden, wodurch der Nutzer nicht auf einen vorinstallierten Interpreter auf seinem System angewiesen ist. Im Nachhineinen gibt es jedoch immernoch die Möglichkeit bei Bedarf auf SWI-Prolog o.ä. zu wechseln.
Wenn man Anfragen an einen Interpreter im interaktiven Modus im Terminal schickt, übergibt man die Anfrage selbst als einen String und als Antwort erhält man ebenfalls einen String. Dieses Verhalten ist gewünscht, wenn man in einem Terminal arbeitet, da man die Ergebnisse in der Regel nicht programmatisch weiterverarbeiten will. Sendet man Anfragen jedoch, wie in unserem Fall, aus Java-Code heraus, will man mit dem Ergebnis einer Anfrage weiter arbeiten. Da wäre es unvorteilhaft, würde man als Antwort vom Interpreter ausschließlich Strings erhalten, da man diese erst selbst in ein gewünschte Format parsen müsste. Solche Parser sind fehleranfällig und aufwändig zu implementieren, bringen aber gleichzeitig dem Projekt wenig Mehrwert. Deshalb sollten man bei der Verwendung des Adpaters bereits verwertebare Objekte als Antwort erhalten, wie beispielsweise *Integer*, *Float* oder selbst definierte Klassen.

3 Interpreterwahl

In Kapitel 2 wurde bereits SWI-Prolog als ein anzubindenden Interpreter fest vorgebeben, sowie die beiden Anforderungen, dass die Interpreter EPL-kompatibel lizenziert sein müssen und außerdem mindestens ein Interpreter in Java implementiert sein muss.
Insgesamt habe ich die Wahl, welche Interpreter angebunden werden sollen, nach folgenden Kriterien getroffen: **(* Pflicht)**

EPL-kompatibel lizenziert* Der Adapter wird Teil eines Eclipse-Plugins sein, welches unter EPL lizenziert ist. Daraus ergibt sich, dass ein verwendeter Interpreter mit EPL kompatibel lizensiert sein muss.
Java-Schnittstelle* Um einen Interpreter von Java-Code aus zu verwenden muss dieser bereits eine eigene Java-Schnittstelle mitbringen. Es wäre zwar möglich auch selber eine solche Schnittstelle zu entwickeln, allerdings liegt das außerhalb des Umfangs dieses Praktikums.
Paket im Maven-Repository Ein Paket, welches im Maven-Repository liegt, kann ohne großen Aufwand in den Build-Prozess eines Projektes eingebunden werden. Dies verringert den Wartungsaufwand und stabilisiert den Build-Prozess über Platformen und Geräte hinweg, da ein Paket direkt aus dem Repository geladen werden kann, anstatt dass nach lokalen Bibliotheken gesucht werden muss. Ein Interpreter, der seine Java-Schnittstelle als Packet im Maven-Repository bereitstellt, verursacht daher allgemein weniger Aufwand bei der Einbindung.
Java-Implementierung Mindestens ein Interpreter sollte in Java implementiert sein, sodass dieser mit dem Plugin zusammen ausgeliefert werden kann. Dadurch muss der Nutzer keinen eigenen Interpreter auf seinem System installieren.
Zeitpunkt des letzten Releases Ein Projekt, welches nicht gewartet wird, bringt ein

gewisses Risiko bei der Verwendung mit sich. Zum Einen werden logische Programmfeheler, die zu falschen Ergebnissen bei Berechnungen führen können, nicht mehr gefixt und zum Anderen könen auch ungepatchte Sicherheitslücken zu einem Problem für den Nutzer werden. Außerdem kann mit der Veröffentlichung neuer Versionen von Drittsoftware die Kompatibilität mit veralteten Projekten gebrochen werden. Daher ist es wichtig, dass ein Interpreter, zumindest zum jetzigen Zeitpunkt, noch aktiv weiter entwickelt wird.

Da eine EPL-kompatible Lizenz und eine bereits vorhandene Java-Schnittstelle Pflichtkriterien sind, werde ich in der folgenden Vorstellung der ausgewählten Interpreter nur in besonderen Fällen weiter darauf eingehen.

3.1 SWI-Prolog

SWI-Prolog[8] ist der am weitestverbreiteste und am aktivsten weiter entwickelte Interpreter. Deswegen wurde in Kapitel 2 bereits vorrausgesetzt, dass dieser Interpreter angebunden werden sollte. Neue Versionen werden teilweise im Wochentakt veröffentlicht, weshalb eine Veraltung des Projektes in nächster Zeit nicht zu erwarten ist. Außerdem stellt SWI-Prolog eine Maven-Paket zur Verfügung, welches allerdings eine Abhängigkeit auf eine native Bibliothek hat. Dies setzt vorraus, das der Nutzer bereits SWI-Prolog installiert und sein System so konfiguriert hat, dass die JVM in der Lage ist, diese Bibliothek auch zu finden.

3.1.1 Installation unter Linux

Zu installierende Pakete: swi-prolog-devel

Nachdem dieses Paket installiert wurde, sollte das Verzeichnis */usr/lib/swipl/lib/* existieren. Je nach Architektur liegt in diesem Verzeichnis ein Ordner *x86-linux* oder *x86_64-linux*, worin wiederum eine Datei *libjpl.so* liegt. Führen Sie folgenden Befehl im Terminal aus: (Beispiel 64-bit Architektur)

```
ln -s /usr/lib/swipl/lib/x86_64-linux/libjpl.so /usr/lib/.
```

Dieser Befehl erstellt eine Verknüpfung der Datei *libjpl.so* im Verzeichnis */usr/lib/libjpl.so*.

3.2 TuProlog

TuProlog[9] ist ein in Java implementierter Interpreter. Er verfolgt die Philosophie, einen möglichst kleinen Kern zu haben und zusätzliche Funktionalitäten über Bibliotheken nachzuladen. Lizenziert ist er unter LGPL, welche im Allgemeinen nicht EPL-kompatibel ist[3]. Jedoch wurde diese Lizenz in der Vergangenheit in einzelnen Fällen doch als kompatibel anerkannt, weshalb es hier eine gewisse Grauzone zu geben scheint. Außerdem befindet sich das Projekt Trust 4.0 momentan noch im Bereich des 'non commercial use', weshalb die Nutzung dieses Interpreter vorerst ohnehin keine Schwierigkeit darstellt. Der letzte Release von TuProlog war im Juli 2019, allerdings nutzte ich in meinem Adapter noch

die Version vom Oktober 2018, da ich mit der Implementierung bereits im Juni begonnen hatte. TuProlog ist außerdem als Paket im Maven-Repository verfügbar, weshalb von Nutzerseite aus keine weitere Installation erforderlich ist.

3.3 Projog

Projog[4] ist ebenfalls ein in Java implementierter Interpreter. Im Gegensatz zu den beiden anderen Interpretern ist dieses Projekt noch sehr jung, wird aber dementsprechend auch noch weiter entwickelt. Ein Nachteil dieser Implementierung ist allerdings, dass sie noch nicht ganz konform zum Prolog ISO Standard ist, weshalb durchaus an einigen Stellen die Funktionalität eingeschränkt sein könnte. Projog steht, wie TuProlog, als Paket im Maven-Repository zur Verfügung und kann somit automatisch durch Maven eingerichtet werden und erfordert keine Installation oder Konfigration nutzerseitens.

4 Prolog4J

Prolog4J[5] ist eine bereits existierende Bibliothek, die verschiedene Prolog Interpreter an Java anbindet. Da der Code unter der MIT Lizenz steht, also EPL-kompatibel ist, kann ich darauf aufbauen und den Adapter so anpassen, dass er den in Kapitel 2 geschilderten Anforderungen gerecht wird. Außerdem ist Prolog4J relativ gut dokumentiert[6].

4.1 Anpassungen

Prolog4J hat bereits Anbindungen an die Interpreter SWI-Prolog, TuProlog, JTrolog, JLog. Damit sind zwei der drei gewünschten Interpreter bereits angebunden und ich muss nur noch die Anbindung für Projog selbst implementieren. Des weiteren wurde das Projekt jedoch seit 2011 nicht mehr aktualisiert, was erfodert, dass sowohl die POM-Dateien als auch der Code aktualisiert wird. Das beinihaltet zum einen, die neusten Versionen von eingebundenen Bibliotheken zu verwenden, und zum anderen eventuelle Änderungen der API solcher Bibliotheken in dern Code einzufplegen. Im letzten Schritt muss noch fehlenede Funktionalität ergänzt werden:

Prologdatenbank aus Datei lesen Um eine Prologdatenbank aus einer Datei zu lesen, kann man die Methode *Prover::loadTheory(InputStream)* benutzen.
Interpreter zurücksetzten Um einen komplett frischen Zustand eines Interpreters zu bekommen, also eine leere Datenbank zu haben, kann man mittels *IProverFactory::createProver()* eine neue Prover-Instanz erzeugen.
OSGI Einbindung Vorhandene Anbindungen an Interpreter sollen nicht statisch im Code verwaltet werden. Stattdessen soll es möglich sein, dynamisch zur Laufzeit zu erkennen, welche Interpreter verfügbar sind. Das ist mit der Laufzeitumgebung von Eclipse, also OSGI, möglich. Damit können Services zur Laufzeit erkannt und eingebunden werden, was in unserem Fall Instanzen von *IProverFactory* sind. In der Klasse *ProverManager* werden verfügbare Anbindungen dynamisch geladen und als List zur Verfügung gestellt.

4.2 Besonderheiten

In diesem Kapitel werde ich einige Besonderheiten erklären, die mir bei der Implementierung aufgefallen sind.

4.2.1 Bug in Projog

Ein Fakt ist ein Ausdruck der Form *human(socrates)*. Dabei ist *human* der Faktenname und *socrates* der Faktwert. Diese Fakten werden in einer Datenbank gespeichert und können auch wieder abgefragt werden. Projog speichert diese Fakten in einer Hashmap und verwendet dabei den Faktennamen (also *human*) als Key und speichert als Wert die Faktenwerte. Fügt man mehrere Fakten mit dem gleichen Namen gleichzeitig der Datenbank hinzu, werden alle Faktenwert gespeichert. Also folgende Anfrage...

```
addTheory(human(socrates), human(euklid)), human(plato)).
```

...resultiert in folgender Datenbank:

```
human(socrates)
human(euklid)
human(plato)
```

Fügt man jedoch alle Fakten einzeln in die Datenbank ein,...

```
addTheory(human(socrates)).
addTheory(human(euklid)).
addTheory(human(plato)).
```

...erhält man folgende Datenbank:

```
human(plato)
```

Es steht also nur noch der zuletzt hinzugefügte Wert in der Datenbank. Das liegt daran, dass anstatt, dass ein Fakt zu den bereits bestehenden hinzugefügt wird, der Wert an der Stelle *human* in der HashMap jedes mal neu überschrieben wird. Gelöst habe ich dieses Problem, indem ich eine eigene HashMap verwalte, und einen neuen Fakt erst dort einfüge und danach alle Werte in der Projog-Datenbank übergebe.

4.2.2 Cons-Operator in SWI-Prolog

Zwei Listen werden in Prolog mit dem sogennaten Cons-Operator konkatniert. Zwei leere Listen zu verbinden sähe beispielsweie so aus: *'.'([],[])*. Wobei . der Cons-Operator ist, und *[]* jeweil eine leere List. Als Ergebnis erhält man logischerweise wieder eine leere List. SWI-Prolog weicht da jedoch etwas vom Standard-Prolog ab, und hat den Cons-Operator durch *[|]* ersetzt[1]. Das heißt, zwei leere Listen zu konkatenieren sieht nun so aus: *'[|]'([],[])*.

4.2.3 Paramtererstezung in SWI-Prolog Termen

Paramter in einer Anfrage werden in der Regel durch Fragezeichen und einen darrauffolgenden Namen markiert, z.B. *is_childof(?parent, ?child)*. Diese Parameter werden dann im nächsten Schritt durch entsprechende Util-Funktionen der Interpreter-API durch tatsächliche Werte ersetzt. SWI-Prolog erwartet allerdings, dass solche Parameter lediglich mit einem einzigen Fragezeichen markiert werden, also *is_childof(? , ?)*, weshalb in *SWIPrologQuery* alle Parameter durch ein Fragezeichen ersetzt werden.

4.2.4 Integer, Long, Float und Double

Alle Interprter unterscheiden zwischen Long und Interger bzw. Double und Float. Der einzige Interpreter, der dies nicht tut, ist Projog. Projog behandelt alle Zahlen entweder als Long oder Double. Um trotzdem noch eine einheitlische Schnittstelle zu allen Interpretern bieten zu können, stehen für alle Interpreter nur noch Long und Double als numerische Datentypen zur Verfügung. Das bedeutet, dass zwar noch Integer und Float als Eingabe gültig sind, aber die Antwort des Interpreters immer zu Long oder Double gecastet wird.

5 Fazit

Das Ziel dieses Praktikums war es, einen Prologadapter für Java zu implementieren. Dazu musste ich erst eine geeignete Auswahl an Intepretern finden. Die Wahl habe ich anhand verschiedener Kriterien getroffen. Bei der Implementierung konnte ich auf die bereits existierende Bibliothek Prolog4J zurückgreifen und musste diese jedoch nach den gegebenen Anfordernungen anpassen und fehlende Interpreter anbinden. Eine gröse Herausforderung war dabei die Aktualisierung des Codes und vorallem der POM-Dateien von Maven.

Literatur

[1] “Cons-Operator in SWI-Prolog”. In: (). URL: https://www.swi-prolog.org/pldoc/man?section=ext-lists.

[2] “Eclipse Plugin Licence”. In: (). URL: https://www.eclipse.org/legal/epl-v10.html.

[3] “EPL-kompatible Lizenzen”. In: (). URL: https://www.eclipse.org/legal/licenses.php.

[4] “Projog”. In: (). URL: http://www.projog.org/.

[5] “Prolog4J”. In: (). URL: https://github.com/espakm/prolog4j.

[6] “Prolog4J Dokumentation”. In: (). URL: https://github.com/espakm/prolog4j/tree/gh-pages.

[7] Robert Heinrich Stephan Seifermann und Ralf Reussner. “Data-Driven Software Architecturefor Analyzing Confidentiality”. In: (2019). URL: https://sdqweb.ipd.kit.edu/publications/pdfs/seifermann2019b.pdf.

[8] “SWI-Prolog”. In: (). URL: https://www.swi-prolog.org/.

[9] “TuProlog”. In: (). URL: https://apice.unibo.it/xwiki/bin/view/Tuprolog/.