TECHNISCHE UNIVERSITÄT DRESDEN

FACULTY OF COMPUTER SCIENCE
INSTITUTE OF THEORETICAL COMPUTER SCIENCE
CHAIR OF KNOWLEDGE-BASED SYSTEMS
PROF. DR. RER. POL. MARKUS KRÖTZSCH

Bachelor's Thesis

for obtaining the academic degree
Bachelor of Science

Integration and Evaluation of an ASP-Solver as an Alternative Reasoning Backend in the Rulewerk Toolkit

Steffen Breuer
(Born 25. September 1996 in Delmenhorst, Mat.-No.: 4682900)

First Reviewer: Prof. Dr. rer. pol. Markus Krötzsch
Second Rewiewer: Dr. David Carral

Dresden, December 16, 2020

Declaration of authorship

I hereby declare that I wrote this thesis on the subject

Integration and Evaluation of an ASP-Solver as an Alternative Reasoning Backend in the Rulewerk Toolkit

independently. I did not use any other aids, sources, figures or resources than those stated in the references. I clearly marked all passages that were taken from other sources and cited them correctly.

Furthermore I declare that – to my best knowledge – this work or parts of it have never before been submitted by me or somebody else at this or any other university.

Dresden, December 16, 2020

Steffen Breuer

Abstract

Rulewerk is a Java API, that is build to abstract the access to data sources for the rule-based reasoner VLog [CDG+19]. Since the VLog reasoner is actively developed, there is always a need to evaluate potential improvements and to compare VLogs performance to other systems. A good way to make it way easier to compare multiple reasoners is to provide them easy access to the same and also diverse data sources. A potential first alternative reasoning backend is the ASP system Clingo [GKKS14]. In this thesis I present a translation form the datalog-based input language of Rulewerk into the ASP-based input language of Clingo. I also show a direct comparison of the performance of both backends.

Contents

1 Introduction

Rulewerk is a Java API, that is build to abstract the access to data sources for the rule-based reasoner VLog To make more use of the capabilities of Rulewerk, this thesis will deal with the integration of the ASP system Clingo v5.4.0 as a possible alternative reasoning backend for Rulewerk v0.7.0. The reason for this integration is not only to make use of the capabilities of Rulewerk, which will make it easier to load different types of datasources, like CSV or OWL files into Clingo, but also to efficiently evaluate changes made in VLog or Clingo. As both systems can load the same data sources, they can comfortably be evaluated afterwards. Therefore the main questions this thesis will answer are: Is it possible to translate the input language of Rulewerk into the one of Clingo, how would an implementation of such a translation look like, how fast is this implementation and how fast is Clingo compared to VLog. To takle all of these querstions is this thesis split into 5 parts and starts first of all with the preliminaries chapter covering the input languages of Rulewerk and Clingo. For this purpose it introduces the respective syntax and semantics of each language. Subsequently the following translation chapter covers the translation of the Clingo syntax into the Rulewerk one. After this is formally done, I will continue with the implementation chapter. Aside from the implementation process itself, it also deals with the problem of loading data from Rulewerk into Clingo and how to get the reasoning results of Clingo back into Rulewerk. Afterwards the forth chapter measures the time it takes to load a knowledge base into VLog or Clingo respectively, followed by a comparison between the reasoning times and finished by the time it takes to load the produced materialisation back into Rulewerk of each backend to evaluate my approach. Finally the last chapter presents some conclusions based on the chapters above and answers the question in what extend it is possible to use Rulewerk as abstraction to load data sources into Clingo.

VLog was build to reason over *Knowledge Graphs* (KG), where this concept is part of the semantic web research field. The KG concept is a knowledge base

2 Preliminaries

My goal is it to integrate Clingo into Rulewerk, but Rulewerk was created to access VLog through Java. Since VLogs input language is based on Datalog with existential rules, does Rulewerk also use this language to handle rules and facts. But Clingo is based on Answer Set Programming (ASP) and therefore there are some syntactic differences I have to cope with and therefore I will discuss in this chapter, what Datalog with existential rules and ASP are and also give an overview about the concrete grammar of VLog/Rulewerk and Clingo.

2.1 Datalog

Datalog is a query language that uses rules to derive new information from given facts.

2.1.1 Syntax

First I will introduce the grammar Rulewerk uses to represent rules and facts. These definitions come from this [MG20] grammar definition. In this grammar literals are called atoms in order "to avoid confusion with RDFLiterals". But the term "RDFLiteral" will only occur once in this thesis and atoms are actually called literals with in Rulewerk and therefore I just changed the name of atoms to literals. But anyways rules and facts are encapsulated in statements

Definition 2.1:
STATEMENT ::= FACT | RULE

Facts consist of a predicate name and a list of at least one ground term. A predicate name itself is either an IRI or a predname. A predname can be any word, that consits of alphanumeric characters, but starts with a letter. A groundterm can be an IRI, a NumericLiteral, or a RDFLiteral.

Definition 2.2:

FACT ::= PREDICATENAME ’(’ GROUNDTERMS ’)’ ’.’

PREDICATENAME ::= IRI | PREDNAME

PREDNAME ::= [a-zA-Z][a-zA-Z0-9]*

GROUNDTERMS ::= GROUNDTERM (’,’ GROUNDTERM)*

GROUNDTERM ::= IRI | NumericLiteral | RDFLiteral

Rules are composed of a head and a body. The head is basically a list of positive literals and the body a list of literals. The only difference between a positive literal and an literal is that literals can also be negated. An literal consists of a predicate name, followed by a tupel of terms. Since Vlog can handle not only negation but also existentially quantified variables there are also existential variables, which only can occur in the head of rules.

Definition 2.3:

RULE ::= HEAD ’⊢’ BODY ’.’

HEAD ::= POSITIVEATOM (’,’ POSITIVEATOM)*

BODY ::= LITERAL (’,’ LITERAL)*

LITERAL ::= POSITIVELITERAL | NEGATIVELITERAL

NEGATIVEATOM ::= ’ ∼ ’ POSITIVEATOM

POSITIVEATOM ::= PREDICATENAME ’(’ TERMS ’)’

TERMS ::= TERM (’,’ TERM)*

TERM ::= VARIABLE | GROUNDTERM

VARIABLE ::= EXISTENTIALVARIABLE | UNIVERSALVARIABLE

EXISTENTIALVARIABLE ::= ’!’ VARNAME

UNIVERSALVARIABLE ::= ’?’ VARNAME

VARNAME ::= [a-zA-Z][a-zA-Z0-9]*

IRI are either an IRIREF or a PrefixedName as defined in RDF Turtle 1.1 [PCBBL14] or respectively the SPARQL Query Language for RDF [PS08]. We will see in chapter 3, that the definition of an IRIREF will be a problem for Clingo and it is defined like this.

Definition 2.4:

IRI ::= IRI_REF | PrefixedName

IRI_REF ::= ’<’ ([^<> "{ }| ^ ‘] - [# x00- # x20])* ’>’

2.1.2 Semantics

In order to derive new knowledge from the given facts we need to know how to handle rules.

Therefore I present the semantics explained by this Lecture [Krö19]

Rules usually contain variables, which need to be replaced with terms.

Therefore we define a gound substitution σ as a mapping from variables to constants.

Let A be an literal and σ a ground substitutuion, then $A\sigma$ is the literal obtained by A by replacing all variables X in A a with $\sigma(X)$ and $A\sigma$ does not contain any variables anymore. The immediate consequence operator T_P is a mapping from a set of facts to a set of facts.

Let I be a finite set of facts and P a finite set of rules, then

$T_P(I) = \{H_1\sigma, \ldots, H_n\sigma | H_1, \ldots, H_n \vdash B_1, \ldots, B_m. \in P \wedge B_1\sigma, ..., B_m\sigma \in I\}$, where $n, m \in \mathbb{N}$, $H_1, \ldots, H_n$ are positive literals, $B_1, \ldots B_m$ are positive literals .

We now use this to define an increasing sequence of databases, where a database is just a finite set of facts. To do so I use a program, which is just a finite set of rules.

Definition 2.5:

Let D be a database and P a program, then

$D_P^1 = D$,

$D_P^{i+1} = D \cup T_P(D_P^i)$ and

$D_P^\infty = \bigcup_{i \leq 0} D_P^i$

Definition 2.6:

soliv Let D be a database, P a program and A some literal, then

A is entailed by $P \cup D$ iff $A \in D_P^\infty$

It is important to note, that first for all $i, j \in \mathbb{N}$, where $i \leq j$ it holds that $D_P^i \subseteq D_P^j$. Since D and P are finite we can only derive finitely many facts i.e. the sequence $D_P^1, D_P^2, \ldots$ is finite and therefore

the must exist some $k \geq 1$, such that $D_P^k = D_P^\infty$. But now it is not so clear, why D_P^∞ is defined as the union of all D_P^i up to infinity. It is defined like this, since we don not know, when to stop appling the immediate consequence operator and therefore the union is just used to count up to infinity.

Negation in datalog brings up some problems, that need to be dealt with. First I explain the general semantics of negation and therefore extend the immediate consequents operator $T_P(I)$, such that $T_P(I) = \{H\sigma | H \vdash B_1, ..., B_n, \sim A_1, ..., \sim A_m. \in P \wedge B_1\sigma, ..., B_n\sigma \in I \wedge A_1\sigma, ..., A_m\sigma \notin I\}$, where $n \in \mathbb{N}_+, m \in \mathbb{N}$.

The first problem I'll point out is that if there is some variable in a negative literal, that does not occur in any positive literal we don't know what domain this variable should belong to. For example the rule $r = \text{happy}(?X) \vdash \sim\text{hasCisis}(?X, ?Y)$ states that X is happy if there is no $\text{hasCrisis}(?X, ?Y)\sigma \in I$ but since we do not have any constrains on $?Y$ we can substitute any groundterm for $?Y$ and could therefore derive unwanted knowledge. To prevent this from happening we want our rules to be safe, which means, that all variables that occur in some negative literal of the rule, also have to occur in some positive literal, which is part of the body of the given rule. Annother problem with negation emerges in the following example.

Example 2.7:

$$\begin{aligned} I &= \{\text{human(adam).}\} \\ P &= \{\text{adult}(?X) \vdash \text{human}(?X), \sim\text{child}(?X)., \\ &\qquad \text{child}(?X) \vdash \text{human}(?X), \sim\text{adult}(?X).\} \end{aligned}$$

Since we do not know if adam is an adult or a child is $T_P(I) = \{\text{adult(adam).}, \text{child(adam).}\}$ and therefore $D_P^2 = \{\text{adult(adam).}, \text{child(adam).}, \text{adam).}\}$. But we now know, that he is an adult and also a child and therefore $T_P(D_P^2)$ is equal to the empty set. Since this is the case, does D_P^3 contain only the fact we have started with. Thus we start all over again and will oscillate between D_P^2 and D_P^3.

To prevent this from happening VLog uses stratified negation, which basically means that the rules are ordered in layers according to their use of negative literals and then those layers are evaluated separately.

Definition 2.8

Let P be a set of rules that contain negative literal and l be a function that maps each natural number to a predicate name that occurs in some rule of P.

l is a stratification of P iff $\forall r = h(t) \vdash a_1(s_1), ..., a_n(s_n), \sim b_1(r_1), ..., \sim b_m(r_m) \in P, \forall i \in$

$\{1, ..., n\}, \forall j \in \{1, ..., m\} : l(h) \geq l(a_i) \wedge l(h) > l(b_j)$

To now use this definition I will redefine the evaluation of databases.

Definition 2.9

Let D be a database, P a program and l a statification of P, where the codomain of l is $\{1, ..., r\}$

- Define sub program for each stratum

  $P_i = \{h(t) \vdash a_1(s_1), ..., a_n, \sim b_1(r_1), ..., \sim b_m(r_m) \in P | l(h) = i\}$

- $D_0^\infty = D$
- for $i \in \{1, ..., r\}$
  - $D_i^1 = D_{i-1}^\infty$
  - $D_i^{j+1} = D_{i-1}^\infty \cup T_{P_i}(D_i^j)$
  - $D_i^\infty = \bigcup_{j \geq 1} D_i^j$
- D_r^∞ is the evaluation of P over D

Rulewerk also allows existential variables in the head of rules, which we call from now on existential rules. Existential rules are quite hard to deal with, since even simple rules can cause an infinit amount of new introduced facts. VLog uses the restricted chase algorithm to deal with these problems. To define the algorithm I again use the first order logic way of writing rules just to make it easier to read.

Definition:

Let D be some Database where $\Sigma \subseteq D$ is the set of existential rules in D and $I \subseteq D$ the facts of D

$\text{chase}(\Sigma, I)$:

$i = 0, \Delta^0 = I$

$\text{while}(\Delta^i \neq \emptyset)$:

$\Delta^{i+1} = \emptyset$

$\text{foreach}(\varphi \rightarrow \exists \vec{z} \psi \in \Sigma)$:

$\text{foreach}(\text{match } \theta \text{ in } \varphi \text{ over } \Delta^{[0,i]}, \text{ with } \varphi\theta \cap \Delta^i \neq \emptyset)$:

$\text{if } (\Delta^{[0,i]} \not\models \psi\theta)$:

$\theta' = \theta \cup \{\vec{z} \mapsto \vec{n}\}, \text{ where } \vec{n} \subseteq \mathbf{N} \text{ are fresh nulls}$

$\Delta^{i+1} = (\Delta^{i+1} \cup \psi\theta') \setminus \Delta^{[0,i]}$

$i = i + 1$

The semantics of existential variables are the only topic of this chapter, which is not presented from [Krö19] but from [Krö18]

2.2 ASP

Answer set programming or in short ASP is a declarative programming paradigm, that is based on logic programs and their answersets.

2.2.1 Syntax

We again differentiate between facts and rules. There are also integrity constrains and quite a lot of other extensions, which can be used in ASP, but since we will not need them I will not present them in depth. Facts are only composed of terms and rules can also contain literals, which basically are just terms which may be negated. Terms themself can be either a simple term or a function. Simple terms can be integers, constants or variables and functions are composed of a starting constant and followed by atleast one term.

Definition 2.10:
FACT ::= TERM (';' TERM)* '.'
RULE ::= TERM (';' TERM)* '⊢' LITERAL(',' LITERAL)* '.'
TERM ::= SIMPLETERM | FUNCTION
LITERAL ::= ('not')* TERM
SIMPLETERM ::= INTEGER | CONSTANT | VARIABLE
CONSTANT ::= ('_')? [a-z] [A-Za-z0-9_']*
VARIABLE ::= ('_')? [A-Z] [A-Za-z0-9_']*
FUNCTION ::= CONSTANT '(' TERM (';' TERM)* ')'

All information about the syntax of Clingo did I gather from [KSW+19]

2.2.2 Semantics

The ASP problem solving process is split into 2 phases: the grounding phase and the solving phase. Clingo itself is only a combination of gringo and clasp, where gringo is a grounder and clasp is the used solver.

Notation 2.11:

Let P be a logic program, which is defined as a finite set of rules over some set A of literals. For some rule $r = a_0 \leftarrow a_1, ..., a_m, \text{not } a_{m+1}, ..., \text{not } a_n$ in P,

where $0 \leq m \leq n$ and each $a_i \in A$ for $0 \leq i \leq n$ This differentiate between the head and the body of a rule just like in Datalog.

$$
\begin{aligned}
head(r) &= a_0 \\
body(r) &= \{a_1, ..., a_m, \text{not } a_{m+1}, ..., \text{not } a_n\} \\
body(r)^+ &= \{a_1, ..., a_m\} \\
body(r)^- &= \{a_{m+1}, ..., a_n\}
\end{aligned}
$$

Rules usually contain variables, which must be eliminated. The grounder systematically eliminates all variables and replaces them with ground facts. A fact is ground, as in first order logic, iff the fact does not contain any variables.

After the grounder is finished and has produced a propositional program, i.e., a set of propositional formulas, this program is passed to the solver which then computes stable models. To understand stable models we first need to define, what it means, that some set of literals X is closed under some positive program P. First a positive program is a program, where $\forall r \in P : body(r)^- = \emptyset$, and X is closed under P iff for all $r \in P$ it is true that if $body^+ \subseteq X$ then $head(r) \in X$.

$Cn(P)$ denotes the smallest set, which is still closed under some positive program P, i.e. for all sets X of literals, that are closed under P, $Cn(P) \subseteq X$ holds. $Cn(P)$ is then called stable model of P. To now extend this to general normal programs I use the Gelfond-Lifschitz Reduct[Gelfond and Lifschitz(1991)] P^X, which is defined as follows. $P^X = \{head(r) \leftarrow body(r)^+ | r \in P \wedge body(r)^- \cap X = \emptyset\}$ Now we can extend the notion of stable models to normal programs, such that X is now a stable model for some program P iff $Cn(P^X) = X$.

All information about the semantics of Clingo did I gather from [Gag19]

3 Translation

In the last chapter we have seen how the grammar of VLog/Rulewerk and Clingo are defined. We have seen that they are actually quite similuar and both use rules and facts for reasoning, but there are still some differences that I need to deal with, in order to use Clingo as alternativ Reasoner. Therefore I will present a translation from Rulewerk syntax to Clingos syntax.
First I will tackle the general syntactic differences and introduce an new function trans. The first difference that needs to be translated is that in Rulewerk syntax all variables starts with a questionmark and it is also allowed to use lowercase letters as the first letter but Clingo does not use questionmarks and the first letter of a variable has to be a capital letter. For this problem I use the subfunction toCaps to make the first letter a capital one. Another problem is that Clingo does not alow IRIs, since constants do not allow the specal characters [^<> "{}| ^ ‘], which are needed to represent IRIs. To solve this I use the function alias, which introduces aliases for each groundterm and functionsymbol in the given database. Since all groundterms and functionsymbols are replaced and constants start with a lowercase letter, but variables with a capital letter I do not need to worry about, that the new aliases might introduce new elements, that were already used in the database.

Definition 3.1:
The domain of syn is the language of Rulewerk and the codomain is defined as the language of Clingo.

$$\mathrm{syn}(x) = \begin{cases} \mathrm{toCaps}(v), & \text{if } x \text{ of the form } ?v, \\ & \text{where } v \text{ is some varname} \\ \mathrm{alias}(x) & \text{if } x \text{ is a groundterm} \\ \mathrm{alias}(f)(\mathrm{syn}(t_1), ..., \mathrm{syn}(t_n))., & \text{if } x \text{ is a literal of the from} \\ & f(t_1, ..., t_n)., \text{ where } n \in \mathbb{N}, \\ & t_i \text{ are terms,} \\ & f \text{ is a predicatename} \\ & \text{and } 1 \leq i \leq n \\ \mathrm{syn}(h_1), ..., \mathrm{syn}(h_n) \vdash \mathrm{syn}(b_1), ..., \mathrm{syn}(b_m). & \text{if } x \text{ is a rule of the form} \\ & h_1, ..., h_n \vdash b_1, ..., b_m. \\ & h_1, ..., h_n \text{ are positive literals} \\ & \text{and } b_1, ..., b_m \text{ are literals} \end{cases}$$

Definition 3.2:

The domain of toCaps is L([a-zA-Z][a-zA-Z0-9]*) and the codomain is L([A-Z][a-zA-Z0-9]*)

$\mathrm{toCaps}(x) = \{\mathrm{V}A \text{ if } x = \mathrm{v}A, \text{where } A \in \mathrm{L}([\text{a-zA-Z0-9}]^*)\}$

Definition 3.3:

alias is a bijection from all groundterms and predicatenames to $\{t_1, ..., t_n\}$, where n is the number of groundterms, plus the number of predicatenames, occuring in a given database.

The next problem is, that Rulewerk allows existentialvariables in the head of rules but existential quantified variables are not expressable in Clingo. To fix this I translate all rules that contain existentialvariables into skolemform, which is the case when a rule does not contain any existentialvariables. To do that I use a new function $\mathrm{toLiteral}$ to construct a new literals a_i for each existential variable, that occur in a given rule $r = h_1, ..., h_n \vdash b_1, ..., b_m.$, where $u_1, ..., u_n$ are positive literals, containing the universalvariables $x_1, ..., x_k$, existental variables $y_1, ..., y_l$ and $b_1, ..., b_m$ are literals containing the universalvariables $z_1, ..., z_t$. Each existentialvariable in r will then be substituted with the corresponding a_i. This correspondence is achieved by initally defining the bijection exNum, which maps existentialvariables to natural numbers, that then are used to name the new introduced predicatenames. Each a_i has all universalvari-

ables that occur in the given rule as it's arguments. This makes shure, that if the body of r is fullfiled, that then a new fact will be added to the database and therefore behaves like existenial variables. To access those universalvariables I use another subfunction $\mathrm{getUniVars}$.

Definition 3.4:

The domain and also the codomain of skol is the language of Rulewerk.

$$\mathrm{skol}(x,r) = \begin{cases} \mathrm{toLiteral}(x,r) & \text{if } x \text{ is a existentialvariable} \\ x & \text{if } x \text{ is a universalvariable} \\ f(\mathrm{skol}(x_1,r),...,\mathrm{skol}(x_n,r) & \text{if } x \text{ is a poritive literal} \\ & \text{of the form } f(x_1,...x_n) \\ \sim \mathrm{skol}(f(x_1,...,x_n),r) & \text{if } x \text{ is a negative literal,} \\ & \text{of the form } \sim f(x_1,...x_n) \\ \mathrm{skol}(h_1,r),...,\mathrm{skol}(h_n,r) \vdash \mathrm{skol}(b_1,r),...,skol(b_m,r). & \text{if } x \text{ is a rule of the form} \\ & h_1,...,h_n \vdash b_1,...,b_m. \end{cases}$$

Definition 3.5:

$\mathrm{toLiteral}(x,r) = f_{\mathrm{exNum}(x)}(x_1,...,x_n)$, where $x_1,...,x_n \in \mathrm{getUniVars}(r)$, $\forall i,j \in \{1,...,n\} : i \neq j \rightarrow x_i \neq x_j$ and $n = |\,\mathrm{getUniVars}(r)|$

Definiton 3.6:

$$\mathrm{getUniVars}(x) = \begin{cases} \{x\} & \text{if } x \text{ is a universalvariable} \\ \emptyset & \text{if } x \text{ is a existentialvariable} \\ \bigcup_{1\leq i\leq n} \mathrm{getUniVars}(x_i) & \text{if } x \text{ is a positive literal} \\ & \text{of the form } f(x_1,...x_n) \\ \sim \mathrm{getUniVars}(f(x_1,...,x_n)) & \text{if } x \text{ is a negative literal,} \\ & \text{of the form } \sim f(x_1,...x_n) \\ \bigcup_{1\leq i\leq n} \mathrm{getUniVars}(h_i) \cup & \text{if } x \text{ is a rule of the form} \\ \bigcup_{1\leq j\leq m} \mathrm{getUniVars}(h_j) & h_1,...,h_n \vdash b_1,...,b_m. \end{cases}$$

Definition 3.7:

exNum is a bijection from all existentialvariables, that occur in a given set of rules to $\mathbb{N}$

In the end I introduce the final translation function trans which now translates a database with rules and facts in Rulewerks syntax into a program in Clingos syntax.

Definition 3.8:

$\mathrm{trans} = \mathrm{syn} \circ \mathrm{skol}$

4 Implementation

In this chapter I will show how I implemented the translation and integrated Clingo into Rulewerk. You can find the code, that is discribed in this chapter under [Bre19]

4.1 General Workflow

First I will explain the general workflow of how to load a logic program into Clingo and then get a stable model back into Rulewerk and afterwards go into the details of each layer. The main problem to solve, besides the translation is, that Rulewerk is written in Java but Clingo is only available as Python, C and C++ APIs we need to pass data from Java to one of those languages and the other way around. I achieve this by using the Java Native Interface (JNI). The JNI allows Java code, that runs in a Java Virtual Machine (VM) to pass and get information from and to applications and libraries, that are written in C, C++ or assembly [JNI docu] and therefore fits exactly my needs. I did write the connection in C++, since there are no critical algorithms that need to be performed but instead I want to improve readability and expandability by using object oriented programming, which is the key difference between C and C++.
To now get back to the workflow, first there are 2 differnt ways of getting rules and facts into Rulewerk in order to reason over them. The first and preferable way is to just loading those rules and facts directly into a knowledge base. The second way, which comes into play if CSV files want to be loaded. In order to do this with the VLog reasoner is able to load them directly into VLog without passing them through Rulewerk, but this is not possible with Clingo yet and therefore they first need to be loaded into VLog, then queried back into Rulewerk from where we then can follow up with the normal workflow. This process only can be done with facts, since there is no way to query rules out of VLog, but CSV files anyway most of the time contain information about facts and therefore this should not be a problem. But now let's get to the general reasoning workflow. I start in Rulewerk were I have defined the ClingoReasoner class, which will be the class in Rulewerk to control the Clingo API. On the other side there is the Control class offered by the C++ API of Clingo. Then I wrote the ClingoControl class on the C++ side which holds an instance of the Control class and is used to make it easier to interact with the Control class. To now bring the ClingoReasoner class and the ControlClingo class together I use the JNI

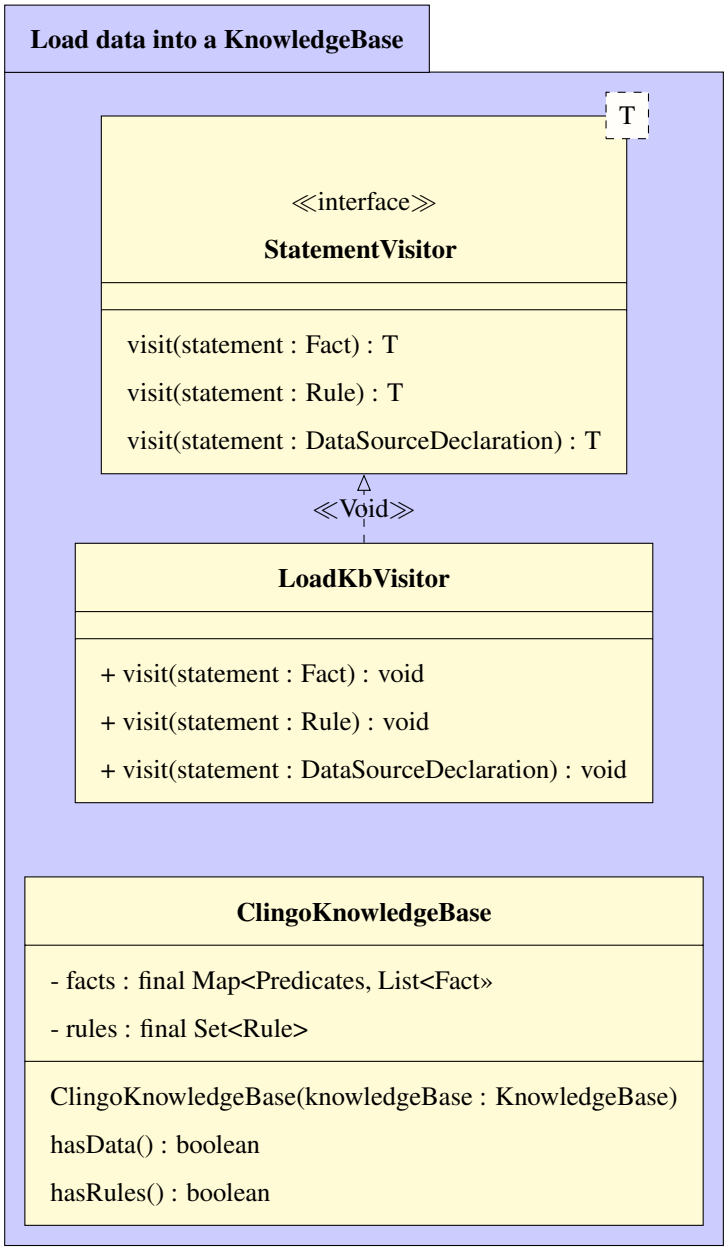
and therefore wrote a Clingo class in Java, which has native methods, which basically are just methods without body which are implemented on the C++ side. From these native methods I then generated a rulewerk_clingo_Clingo file which contains corresponding methods to the native methods of the Clingo class. An instance of the ClingoReasoner will then hold an instance of the Clingo class of the Java sided JNI layer and the C++ sided rulewerk_clingo_Clingo file holds an instance of the ClingoControl class. Now the ClingoReasoner class and the Control class of the Clingo API are connected and I am able to pass rules and facts from Rulewerk to Clingo and the other way around.

4.2 Rulewerk

Now I will go deeper into Rulewerk side. As we have seen in the general workflow the ClingoReasoner class is the main class to use Clingo. But there still is a lot of things, that needs to be taken care of, which are not directly related to communicating with Clingo.

4.2.1 ClingoKnowledgeBase

The first task that needs to taken care of is, that rules and facts need to be hold in a structured way to access them and pass them to Clingo. For this purpose I created the ClingoKnowledgeBase class. The KnowledgeBase class has a Set of rules that is implemented as a Hashset<Rule> and a Map that maps predicates to a List of Facts and is implemented as a HashMap<Predicate, List<Fact». Both of those datastructures are from the Java Collections Framework. To fill those structures I used the visitor design pattern, to avoid unnecessary type casting. The main idea of this pattern is that instead of directly adding the rules and facts to their datastructures there is an extra visitor class, in my case the LoadKbVisitor class, that has different visit methods that are overloaded and therefore can handle rules and facts differently. Rulewerk even provides a StatementVisitor interface that I just implemented for my purposes in the LoadKbVisitor class.



4.2.2 Translate Statements into Clingo syntax

Now that we know how to get the rules and facts into the ClingoKnowledgeBase, I will explain to get them out. To handle them easier within Rulewerk, the ClingoKnowledgeBase holds its data still in Rulewerks syntax but since I want to pass them into Clingo I still need to translate them into Clingos syntax. In chapter 3 we have seen, that translation consists of 3 phases first the general syntactic differences, then the replacement of all predicate names by aliases and finally the skolemization of existential rules. In order to solve the general syntactic differences I wrote the ModelToClingoConverter class. Since the Skolemization and the replacement of all predicates by some aliases require an inner state they both get their own class: the Skolemizer and the AliasHandler class. Since the general syntactic differences do

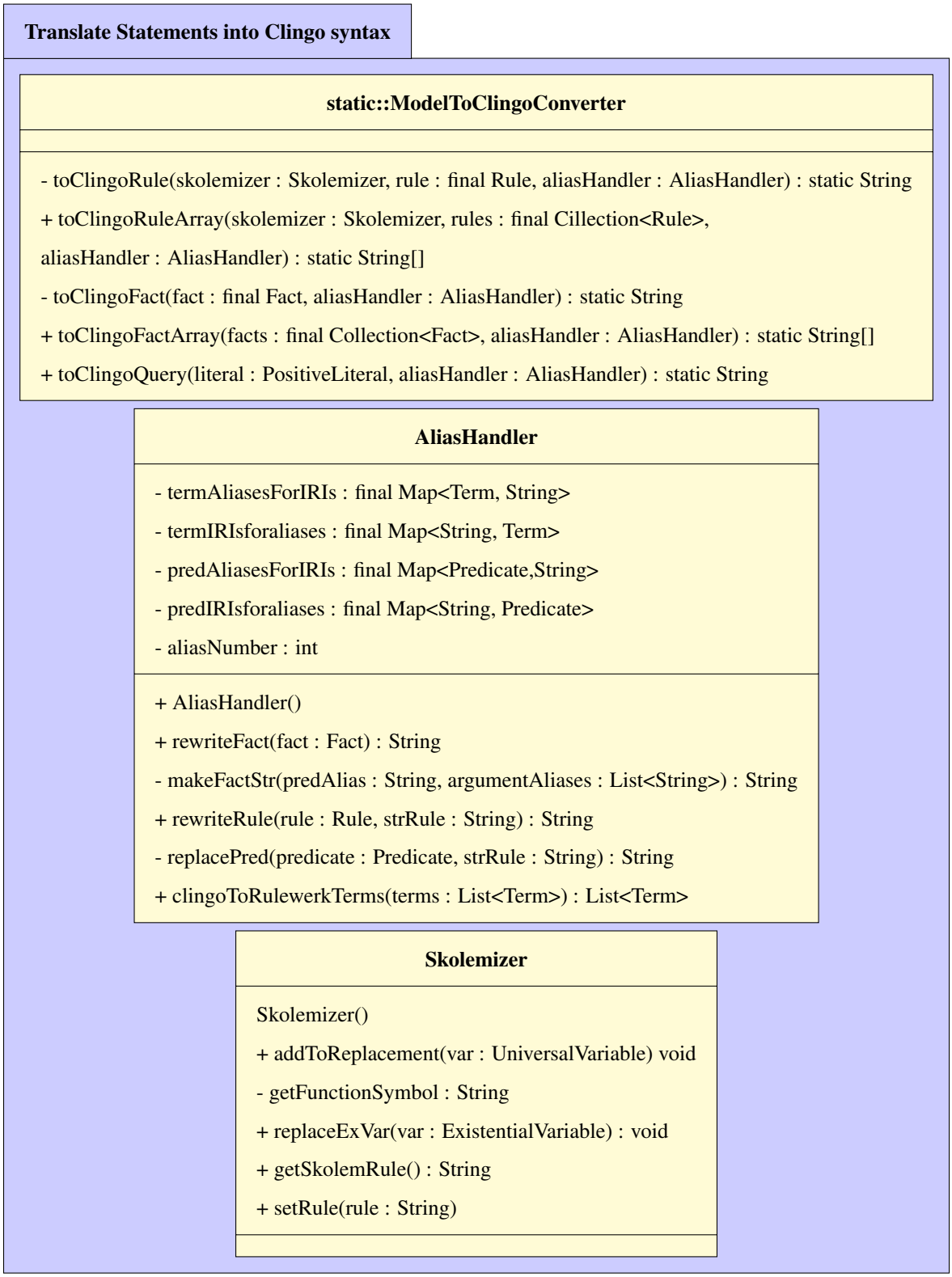
not need an inner state to be handled all the functions in the ModelToClingoConverter class are static, it does not even hold an instance of the Skolemizer or AliasHandler since both classes are also needed to translate the resulting stable model, that comes back from Clingo and therefore are needed in the ClingoToModelConverter class, which I will introduce later.

The ModelToClingo class contains a toClingoRuleArray and a toClingoRule method, where the toClingoRuleArray method just takes care of handling an Array of rules and the toClingoRule method does the actual translation. As we will see later on Clingo takes strings as input and therefore the translation starts with using the getSyntacticRepresentation function provided by Rulewerk. Then all the questionmarks, from the universal quantified variables get replaced by an empty string. After that, if the rule contains existential variables, it uses an instance of the Skolemizer class, that it gets passed as an argument, to replace all existential variables with a fresh predicate, that has all universal variables as arguments. The toClingoRule method first uses the addReplacement function of the Skolemizer to add all universal variabels to a string that will be used as a body for all functions, that will replace the existental variables. After that all the existential variables get replaced with a fresh function by the replace function, that uses the getFunctionSymbol function of the Skolemzier, that is used to construct a fresh function symbol by using an integer functionCount, that is hold in the Skolemizer and incremented after every predicate construction. When this process is finished the skolemized rule gets returned and the state of the used body gets reset. After the skolemization, all predicates get replaced with aliases by the AliasHandler, by using the rewriteRule function. This function gets the already skolemized rule, where the replacements are being performed on and the inital rule to easily access all predicates, that need to be replaced. The rewriteRule function replaces first the head predicates and then the body predicates, by using the replacePred function. The AliasHandler holds the map predAliasesForIRIs, that maps Predicates to there corresponding alias, which are stored as a String. This is nessesary since all predicates with the same name should obviously be replaced with the same alias. Therefore the replacePred function checks first if a given predicate is already in the keys of predAliasesForIRIs and if this is the case it just replaces the names of those predicates, which can be accessed by the getName function, with the alias that is stored in predAliasesForIRIs. To achieve this I use the replace function that is provided by the String class. A problem that occurs is, that IRIs within rules are surrounded by left and right angle brackets, but the names of those IRIs do not contain those brackets. In addition those bracket are not part of the Clingo grammar and therefore have to be erased. If a given predicate is not already contained in predAliasesForIRIs a new alias needs to be constructed. This alias is constucted by a string "pred" concatinated with an integer, that is hold by the AliasHandler and incremented after each construction, then the given predicate name is replaced by the new alias and finally the alias is added with the given predicate as key

to predAliasesForIRIs and also a map predIRIsforaliases which maps the predicate and the alias the other way around, which will be needed in the backtranslation process of queries.

The ModelToClingo class also contains a toClingoFactArray and a toClingoFact method. The toClingoFactArray method is again as the toClingoRuleArray method just used to handle arrays and the toClingoFact method does the actual translation, but this time it is way simpler than for rules, since facts do not contain any variables, that have to be replaced and also don't contain existential variables. Therefore both methods have only the AliasHandler as arguments and of course an array of facts or respectively a single fact. The rewriteFact method just calls the rewriteFact method of the AliasHandler. This method operates quite differently, than the rewriteRule method, since a fact does not only contain a predicate that need to be replaced, but also terms that can be IRIs. So basically every part of a given fact except for the brackets and the dot in the end of the fact have to change. But this is very inefficent and therefore I just construct a new String from the given fact by using the makeFactStr method. So first I collect the alias for the predicate of the fact and either get it again from predAliasesForIRIs map or if the given fact is not yet contained in it I construct a new one in the same fashion as in the rewriteRule method. The arguments of facts are Terms and since Terms and Predicates do not have a common upper class I need a new map to store their aliases, which is called termAliasesForIRIs and maps terms to strings. The same time also a termIRIsforaliases map gets filled, that maps Strings to Terms, which will be needed to translate back to The input terms, which will be needed later. Anyways the construction of new aliases follows the same pattern as for predicates except for the dummi String, where I concatenate aliasNumber integer to. I use const as dummy string to make it easier to debug the program. Then the alias of the predicate and a list of aliases of the argument terms get passed to the makeFactString function, to concatenate these strings together and make it a fact in Clingo syntax.

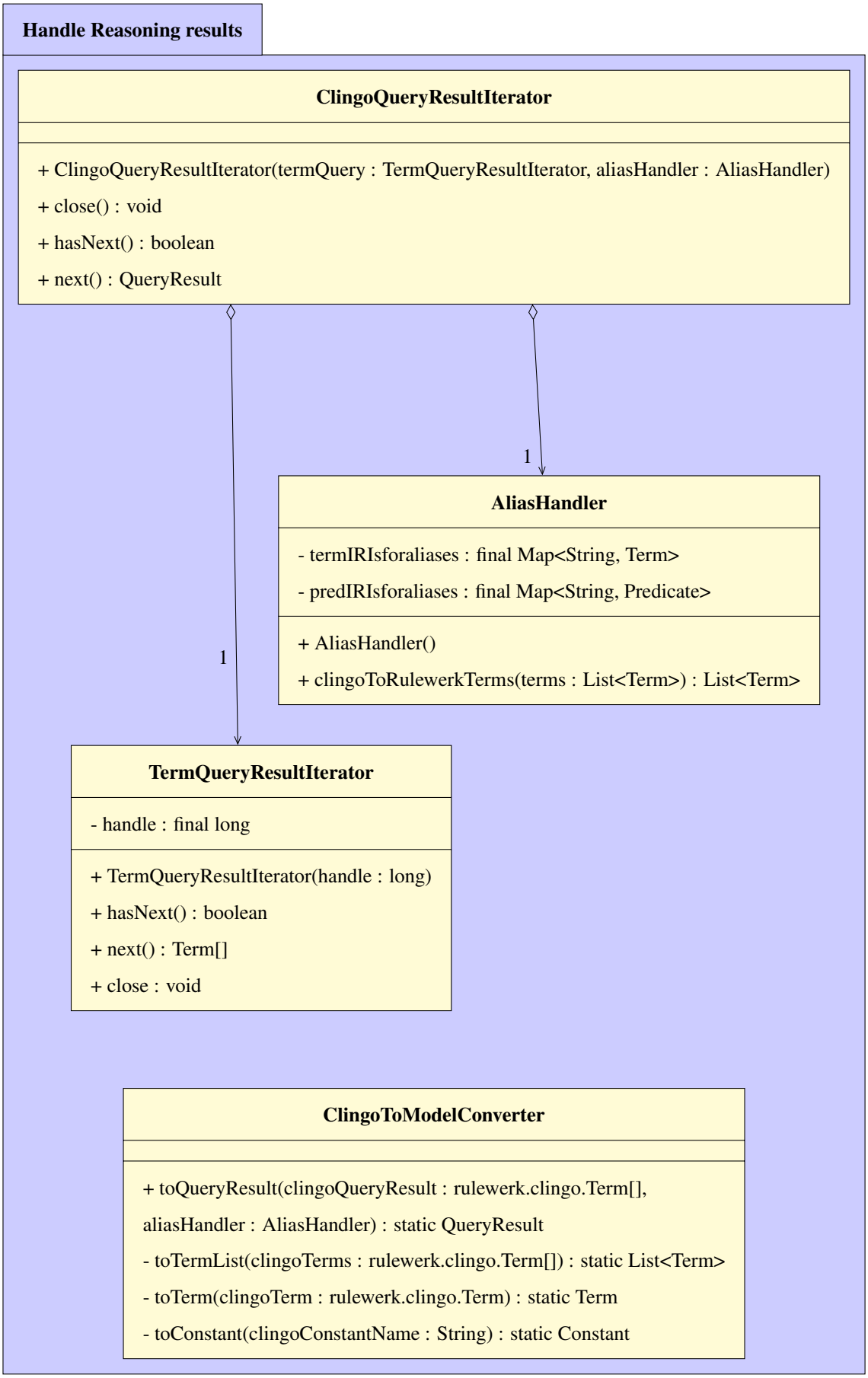
But rules and facts are not the only things, that need to be translated. Within Rulewerk, we work with its own syntax and the ClingoKnowledgeBase also stores its facts and rules like this. But in order to query all predicates that occur in the knowledge base, I also need to translate queries into rule syntax. For this purpose the ModelToClingo class contains the toClingoQuery function. Since a query basically just asks for all ground terms in rulewerk syntax or constants in Clingo syntax, that belongs to a given predicate. Therefore the predicate, that is queried for is just replaced by its alias.



4.2.3 Handle Reasoning results

Now we have finshed the part of the rulewerk-clingo package were the Rulewerk syntax is translated into Clingos syntax and move on how to handle the results of queries and translate them back into Rulewerk

syntax. In order to query a particular model the Clingo reasoner has produced, I use a ClingoQueryResultIterator, iterates over the answertupels of some particular predicate. First the ClingoQueryResultIterator class gets passed an instance of the AliasHandler and also from the TermQueryResultIterator which is an iterator class belonging to the Java side JNI layer. To now iterate over the resulting model the Clingo reasoner has produced, the interator uses its next method, which first calles the next method of the TermQueryResultIterator class. This method returns an array of terms, where this term class is also part of Java side JNI layer. Then the toQueryResult method of the theClingoToModelConverter class gets used, to produce a QueryResultImpl instance. The QueryResultImpl class is provided by Rulewerk and Implements the QueryResult interface. The QueryResultImpl provides some additional functionality to a list of facts, like an equals function. But anyways, the toQueryResult starts with transforming this array JNI terms into a list of Rulewerk terms using the toTermList method. Since this list still contains the aliases, that the input terms where asigned to, the next step is, to use the clingoToRulewerkTerms method of the aliashandler to transform those back into the Terms we have started with. To do this the clingoToRulewerkTerms uses the map termIRIsforaliases, which holds all the corresponding predicates of the aliases, that are contained in a queried Terms, to return the correspendent of a particular alias, which the method got as argument. Finally a QueryResultImpl object gets constructed by using the translated list of terms, which then gets returned. But the next method is not the only method of the ClingoQueryResultIterator class. It also contains a hasNext method and a close method, which both just run the corresponding methods of the TermQueryResultIterator instance.


Handle Reasoning results
ClingoQueryResultIterator
+ ClingoQueryResultIterator(termQuery : TermQueryResultIterator, aliasHandler : AliasHandler)
+ close() : void
+ hasNext() : boolean
+ next() : QueryResult
1
1
AliasHandler
- termIRIsforaliases : final Map<String, Term>
- predIRIsforaliases : final Map<String, Predicate>
+ AliasHandler()
+ clingoToRulewerkTerms(terms : List<Term>) : List<Term>
TermQueryResultIterator
- handle : final long
+ TermQueryResultIterator(handle : long)
+ hasNext() : boolean
+ next() : Term[]
+ close : void
ClingoToModelConverter
+ toQueryResult(clingoQueryResult : rulewerk.clingo.Term[],
aliasHandler : AliasHandler) : static QueryResult
- toTermList(clingoTerms : rulewerk.clingo.Term[]) : static List<Term>
- toTerm(clingoTerm : rulewerk.clingo.Term) : static Term
- toConstant(clingoConstantName : String) : static Constant

4.2.4 ClingoReasoner

Finally I will explain how the ClingoReasoner class wraps those functionallities to use them in Rulewerk and it implements the Reasoner interface provided by Rulewerk. There are 3 main tasks that the reasoner fulfills and in order to deal with them there are serveral objects needed, which are instantiated in the constructor of the Clingo reasoner. We will need an instance of the Clingo class which will be explained in the Java Native Interface section, but it basically alows to control Clingo within Java, then we will need an instance of the AliasHandler and the Skolemizer. In order to ensure a consistent state within Clingo the reasoner also owns a ReasonerState enum, which initally is set to KB_NOT_LOADED and finally also a static instance of the Logger class, which is provied by Rulewerk. Now I go on with the first task, which is holding a knowledge base in order to operate on it and therefore gets passed one on construction. The next task that the reasoner needs to take care of is obviously reasoning. Therefore it has a the reason function, which takes care of loading the knowledge base into Clingo and also start the reasoning process. In order to load the knowledge base the reasoner has an load function. This function decides by evaluating the reasoner state what to do. When the state is set to KB_NOT_LOADED it runs the loadKnowledgeBase method which I cover next. When the current state is KB_CHANGED then first the reasoner gets reset by using the resetReasoner method, which just calls the stopSolver method of the clingo object, then sets the reasoner state back to KB_NOT_LOADED and loggs, that the reasoner was reset. The loadKnowledgeBase method now first constructs a ClingoKnowledgeBase, then starts the clingo reasoner using the startSolver method of the clingo object. After that the loadFacts and loadRules methods, which get passed the ClingoKnowledgeBase, and finally the reasoner state is set to KB_LOADED. The loadFacts method then takes the facts from the knowledgebase and uses the static toClingoFactArray method of the ModelToClingoConverter class to convert the facts into clingo syntax and here comes the aliahandler into play since the toClingoFactArray method takes it as an argument. The converted facts then get passed into clingo using the addStatement method. The loadRules method follows the same pattern but it uses the toClingoRuleArray function which additionally to the allias handler also the skolemizer. Now let's get back to the reason method. Now in case the knowledge base is loaded into clingo the runClingo method is called, which calls first the ground and then the solve method of the clingo object. After that the reasoner state is changed to MATERIALISED and finally it returns true. The final task now is to get the results back into Rulewerk. In order to do that I overwrote the answerQuery method which comes from the Reasoner interface which takes a PositiveLiteral and also a boolean. The boolean is supposed to determine weather nulls should also be queried, but this is useless for clingo and therefore it is ignored. The answerQuery method, from the ModelToClingoConverter then turns the PositiveLiteral into Clingo syntax using the toClingoQuery, additonally to the PositiveLiteral

the aliasHandler. Now this clingo PostiveLiteral, which acctually should be called term now is passed to the termQuery method of the clingo object, which creates a ClingoQueryResultIterator which then gets returned. Now we know how to query for a single PositiveLiteral, but most of the time we want to query for all predicates within the knowledge base. Therefore I wrote the getPosLitToQuery method which first gets all predicates, that are contained in the knowledge base and then turns them into PositiveLiterals by using the predToPosLit method.

4.2.5 CSVloader

As we have seen in general workflow, CSV files, which usually contain facts, get loaded into VLog as in memory data sources, but in order to load these facts into Clingo, those files need to be loaded back into Rulewerk. To take care of this task I wrote the CSVloader class. This loader assumes, that all csv files are GNU zip compressed and contained in a single folder and that there is some sort of config file, which contains for each CSV file a line that contains first the name of the predicate the groundterms in the corresponding CSV file belong to, then follows the name of the file the line corresponds to and finally the arity the predicate has.
To now fill a given knowledge base with the facts of a given folder first this knowledge base gets passed in constructor of the CSVloader and also a new knowledge base gets constructed, which will be used to load the in memory data sources into VLog. I use two different knowledge bases since I do not want to load rules and facts, that might already exist in the passed knowledge base, into VLog. Then a VLogReasoner gets constructed, thats used to load the facts. To now actually load all CSV of a folder the CSVloader has the method loadCSVsfromFolder, which takes two strings. One for the path of the config file and the other one for path of the folder the CSV files are stored in. The method then first uses the getLoadInfo method to get an list of arrays of strings where each string in the array contains one of the parts of the config file. Then it iterates over the list and composes a string which then can be parsed by the RuleParser into a DataSourceDeclaration and adds it to the knowledge base, which is used to load these into the reasoner. then the reason method of the reasoner is called and afterwards the forEachInference method of the reasoner is called to load the facts into Rulewerk. This method takes a InferenceAction, which determines what to do with the query results it queries. Therefore let the CSVloader be an implementation of the InferenceAction interface and added a visit function which then adds the results to the result knowledge base.

4.3 Clingo

4.3.1 ControlClingo

Now I will explain how the interaction with the Clingo API works and what I did to make it easier to interact with it. The main class is the ControlClingo class. The main task this class has to deal with is adding rules and facts into Clingo and also control the grounding and solving process. On construction of an ControlClingo instance a Control object, that is provided by the Clingo API is created and hold by the instance. This object will solve all of the tasts ControlClingo has to cope with. First I will explain how to get rules and facts into Clingo. Clingo is able to solve logic programs in parts. This is used for multishot solving (A Tutorial on Hybrid Answer Set Solving with clingo), but since Rulewerk is not build to deal with continuouslly changing data, it won't be very usefull. But still a part has to be specified to add data to the solver. To do this a dummi part is set in the constructor of the ControlClingo class, which will be used for all interactions with the solver. After the part is set it now is possible to add rules and facts to this part and in order to do that it is just one method needed and it is called add addStatement. In the Clingo grammar rules and facts are not encapsulated by the term statement, but in order to make it easier to talk about them I will now just call them statemnts. To understand why only on method is needed let's take a look at how the Control class adds statements to the reasoner. The Control class just has a method add which takes a C-style string(const char[]), which defines which part the given statement is added to, then it also takes a StringSpan which boils down to an constum made iterable type, that iterates over c strings. This StringSpan is used for addtional parameters, that can be added to the part, that the new statements are added to. But since we only use one part those parameters won't be very usefull. The last argument is finally the statement also as C-style string. the addStatement method now wraps this up by just getting the statement as argument and passing the dummi part that is stored in the instance of the ControlClingo object, adding an empty StringSpan as arguments and finally passing the statement, that it gets as an argument to the add method.

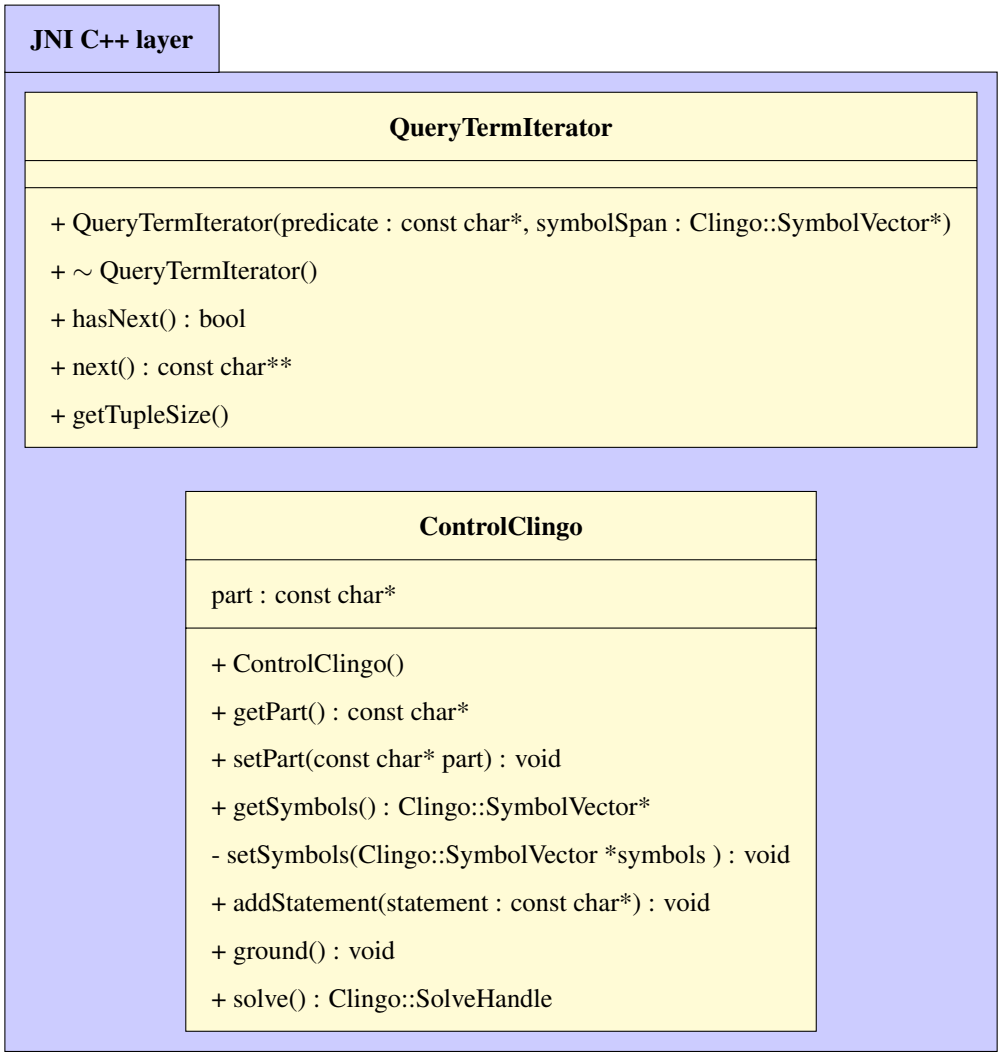
The next step is to ground a program. In order to do that the Control class of Clingo has a ground method, that takes a PartSpan, which again boils down to a iterable but this time of Part objects, which are also privided by the API. Therefore the ground method of the ClingoControl class constructs a Part object, from the part, which is stored in the ControlClingo class, adds it to a new PartSpan and passes it to the ground method of Clingo.

Finally we deal with the solving part. The Control class of course also offers solve method. This method only takes a LiteralSpan, which is used to add assumptions to the solving process, which means, that we can pass some literals to the solver which are not part of our program, but we think that they are supposed to be true. Since Rulewerk just wants to operate on the knowledgebases, this option is not really useful

for now. Therefore does the solve method of the ClingoControl object just pass an empty Literal span to the solve method of the Control class. The solve method of the Control object returns a SolveHandle object, that can be used to access the stable models. The stable model semantic of ASP allows to have more that one model to a given program and all of them need to be safed. The Solvehandle object has a next method, just like an Iterator. the solve method of the ClingoControl class then uses the saveSymbols method, to safe all models as a vector of symbol vectors.

4.3.2 QueryTermIterator

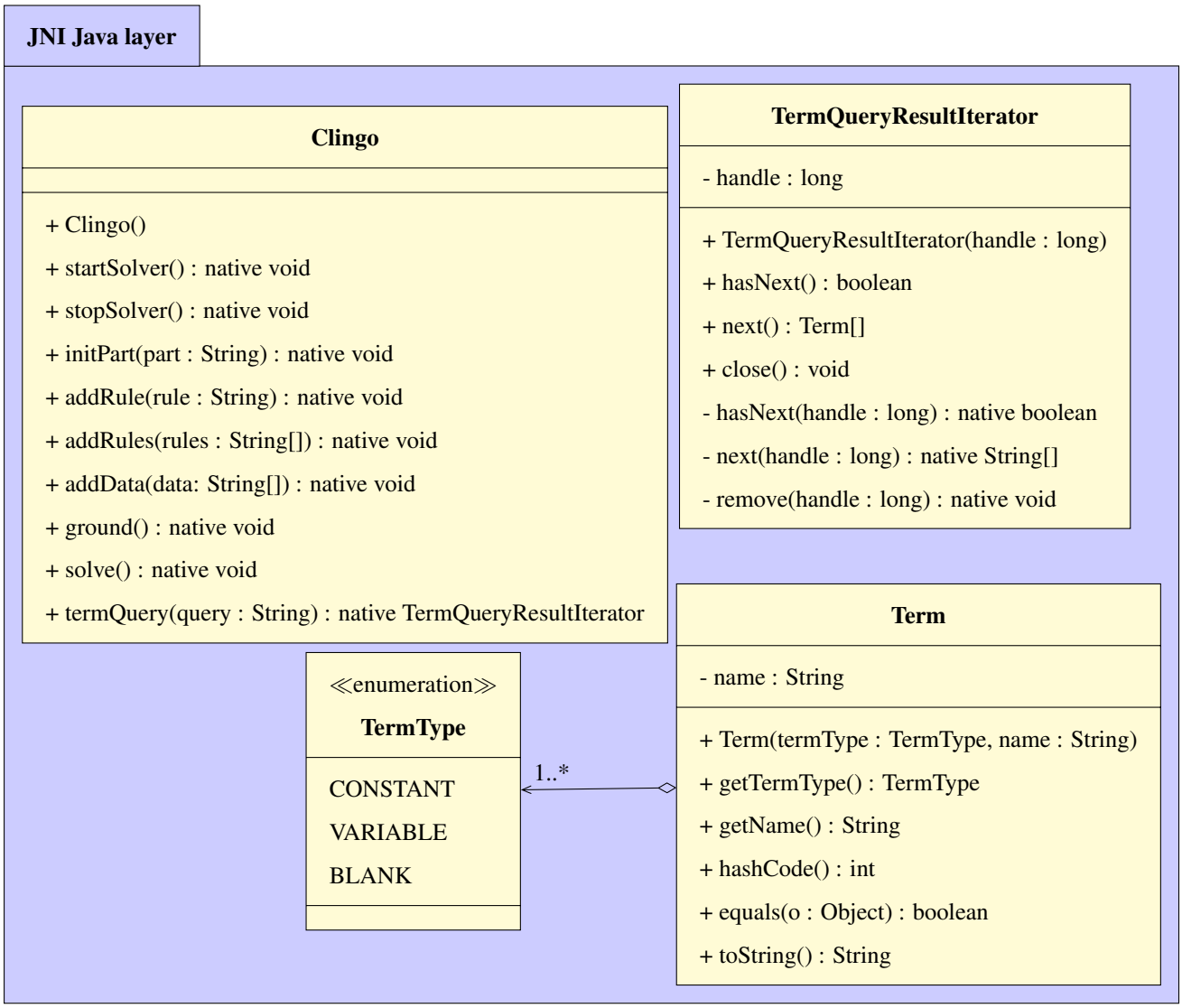
Now that we know how to get a program into Clingo and ground and solve it, its time to explain how to answer queries on those models. For this purpose I wrote the QueryTermIterator class. As we have seen a query asks for all instances in a given model for some predicate. And Therefore the constructor of the QueryTermIterator class gets passed the predicate, which is asked for and also the given model as SymbolVector. From the SymbolVector a iterator is saved, which is provided by the standard library Vector class. The constructor also uses the findNext method to iterate over the SymbolVector, to find the first symbol, where the name of the symbol matches the predicate that is queried for. Since the QueryTermIterator is an iterator class it of course contains a hasNext method and a next method. The hasNext method just checks if the iterator points to the end of the symbol vector. The next method first constructs a array of C-style strings, which is filled with the arguments of the symbol, that is hold in the position of the SymbolVector, where the iterator points at and then also uses the findNext method to set the iterator to the next Symbol, which arguments will be returned, the next time the next method is called.



4.4 Java Native Interface

Now that we know how the Rulewerk and the Clingo side work I will explain how the Java Native Interface (JNI) operates in order to connect them. The JNI introduces the keyword native. This key word alows to define native methods, that have an empty body. From those methods the java compiler can then create a C++ header-file which contains corresponding methods. The methods in this header file can then be implemented and get called when ever the native methods are used. Java and C++ use different datastructures even to store something as simple as strings. To translate those structures every implementation of native methods get passed a JNIenv object which can cope with those problems and contains methods to translate those structures even for custom made objects. Let's now jump into the actual JNI layer I wrote. I will explain the Java and C++ simustaneously since the Java side almost only contains native methods, that are anyways implemented on the C++ side. First the Clingo class, that was already introduced in the general workflow. This class basically mirrors the ControlClingo class since it is just

used to control it through java. It contains a startSolver and endSolver method. The startSolver method is used to construct a ControlClingo object, which is then held on the C++ side indefinitely and in order to delete this object when the solving prosses is done and free the used space the stopSolver method is used. Of couse it also contains a addStatement method, which takes an array of strings. Its implementation converts this array into an array of C-style strings which then gets passed to the addStatement method. Then there is also a ground and a solve method, which just call there corresponding methods of the ControlClingo class. and finally a termQuery method that takes a string, that indicates which predicate is queried for and an integer, that defines which of the models is used to perform the query on. The implementation then just converts the string into a C-style string and the integer into a C++ integer and passes it to the termQuery method, which then returns a TermQueryIterator object. This is possible since I also defined a TermQueryIterator class in the JNI layer. The constructor of this class get passed a long, that contains a pointer to the TermQueryIterator object, that was discussed in the Clingo section. The JNI layer version of this object contains a normal hasNext method which calls a native version of the hasNext method, that additionally gets passed the pointer to the Clingo side TermQueryIterator and then calles its hasNext method. The next method of the Java side TermQueryIterator also calles a native version of the next method, that again passes the pointer, but the native next method returns a array of string which then get transformed into an array of Terms. In order to perfom, this I wrote a Term class on the java side of the JNI layer, which exactly looks like the definition in Rulewerk and therefore can be used within Rulewerk.


JNI Java layer
Clingo
+ Clingo()
+ startSolver() : native void
+ stopSolver() : native void
+ initPart(part : String) : native void
+ addRule(rule : String) : native void
+ addRules(rules : String[]) : native void
+ addData(data: String[]) : native void
+ ground() : native void
+ solve() : native void
+ termQuery(query : String) : native TermQueryResultIterator
TermQueryResultIterator
- handle : long
+ TermQueryResultIterator(handle : long)
+ hasNext() : boolean
+ next() : Term[]
+ close() : void
- hasNext(handle : long) : native boolean
- next(handle : long) : native String[]
- remove(handle : long) : native void
≪enumeration≫
TermType
CONSTANT
VARIABLE
BLANK
1..*
Term
- name : String
+ Term(termType : TermType, name : String)
+ getTermType() : TermType
+ getName() : String
+ hashCode() : int
+ equals(o : Object) : boolean
+ toString() : String

5 Evaluation

Now that we have seen how the integration of Clingo is done I will present some tests I did to evaluate my approach and also to find weak points, that need to be improved.

5.1 Dataset

I use a the Lehigh University Benchmark (LUBM) [GPH05] dataset, which also was used in (VLog paper) to evaluate VLog. You can find this dataset here [Bre19]. LUBM is a common benchmark, which describes Universities. This dataset is split into two parts: a TBox, which is stored as an OWL file and an ABox, which is stored as a GNU zip compressed CSV file. Those terms are widely used when building ontologies. An ABox contains all concept assertions and propery assertions. Concept assertions and property assertions are terms, that come from the description logics field and in our case correspond to facts, where classes correspond to predicates of arity one and properties correspond to predicates of arity two and from now on I just call them facts. A TBox contains all the abstract information, that is given and usually expressed as General Concept inclusions or property inclusions, which again are a terms from description logics and can be expressed as rules in datalog and from now on I will also just call them rules.
The TBox of LUBM is fixed and contains manually created rules, which are expressed in OWL Lite and therefore contains restrictions like $\mathrm{inverseOf}$, $\mathrm{TransitiveProperty}$, $\mathrm{someValuesFrom}$ and $\mathrm{intersectionOf}$. Important here is, that rules containing the $\mathrm{someValuesFrom}$ restriction get parsed into existential rules, when parsed into Rulewerk and therefore this dataset fits perfectly to evaluate my aproche since the skolemization of existential rules also gets tested.

ABoxes can be manually created and also uses the OWL lite language to express its facts. I used two

# Rules	# Univeral rules	# Existential Rules
96	88	8

Table 5.1: TBox specs

ABox	Classes	Properties	facts
1	14	12	2283599
2	14	12	4549977

Table 5.2: ABox specs

different ABoxes. In the following table are the general specs of those ABoxs listed

5.2 Evaluation setup

Now will show how the tests where performed. First all tests where run on a Lenovo ideapad520S with 8 GB of RAM and an Intel Core 2,5GHx2 i5-7200U prozessor. In order to test my approach I run Clingo on both datasets and compared it with VLog in two different setups. To measure the times each approach takes I used the Stopwatch class from com.google.common.base.Stopwatch. I also double checked the times I measured for the reasoning part of VLog, by using the VLog Logger and also the loading time combined with the grounding and solving time for Clingo by using the statistics provided by the Clingo control class. Let's get to the setups of each test.

I start with the setup of the test for Clingo. As we have seen in the general workflow, in order to load GNU zip compressed CSV files into Clingo, first the ABox needs to be load, as in memory data source, into VLog and then all facts get queried back into Rulewerk. Then the TBox gets loaded by using the OWLOntologyManager and then the OwlToRulesConverter converts the OWL TBox into Datalog rules, that get passed into a knowledge base object. Both classes are provided by Rulewerk. This complets the first part of the test, which gets measured. The next part, that is measured is the loading of the knowledge base into Clingo, which involves the conversion of all rules and facts into Clingo syntax. After that the reasoning time gets measured. Since I am not interessted in the time it takes to ground and solve separately, I did not measure them separately. For the solving part I used that standard settings and single thread solving. As we have seen in the semantics part of Clingo, we know, that Clingo can produce more than one model but I restricted this in order to compare it with VLog to just one model, that is searched for. The final part that is measured is the loading of the resulting model back into Rulewerk. I just used the ClingoQueryResultIterator to iterate over all results without adding them into a knowledge base, since I really want to measure the difference it takes to convert the rules back into Rulewerk syntax and don't want to measure unnessesary operations. And finally I also measured the overall time the test took.

run	whole run	loading from File	loading into Clingo	solving	querying
1	87.568	16.102	25.673	23.334	22.442
2	88.116	16.206	25.777	23.578	22.526
3	87.991	16.296	25.675	23.557	22.439
AVG	87,892	16,201	25,708	23,490	22,469

Table 5.3: Clingo times in seconds for dataset 1

Next I present the first setup for the test of VLog. This setup follows the pattern of the Clingo setup. obviously it is unnessasary to first load the ABox as an in memory data source into VLog and then back into Rulewerk and then again back into VLog. But with this first setup I wanted to achieve a comparison for the loading and querying time and therefore this loading pattern is necessary. For the reasoning in this approach I used the the restricted chase algorithm, to get peak performance out of VLog
The second setup for the Clingo now does not load the ABox back into Rulewerk and therefore I summed the loading time that it takes to load the TBox into VLog and the time it takes to prepare the in memory datasource into one measurement. The solving and querying is still meassured separately as in the other appoaches. This setup is used to see the impact of just loading the in memory datasource on the overall time difference between using Clingo or VLog as reasoners for Rulewerk.
I run those 3 setups with each of the 2 datasets 3 times, to avoid to present outliers, which would miss-represent the results.

5.3 comparison

Now I will take a look at the results and point out some interesting findings. Let's first take a look at the differences in loading the knowledge bases into Clingo or respectively VLog. Here we see, that the difference for the first dataset is actually quite huge, but when we take a look at the second dataset, we can see that Clingo is actually around 3 seconds faster in loading the dataset. But when we compare this to the second setup VLog, we see that the whole loading process take way less time. When we take a look at the solving times, we see that VLog again is way faster. The same is the case for querying the data back into Rulewerk.

run	whole run	loading from File	loading into Clingo	solving	querying
1	193.901	29.283	61.044	55.004	47.235
2	194.957	29.577	61.268	55.257	47.500
3	196.040	29.823	61.456	55.914	47.494
AVG	194.966	29.561	61.256	55.392	47.410

Table 5.4: Clingo times in seconds for dataset 2

run	whole run	loading from File	loading into VLog	solving	querying
1	30.799	16.358	7.592	1.045	5.383
2	31.395	14.836	8.870	1.433	5.848
3	30.631	14.777	8.703	1.426	5.316
AVG	30.942	15.324	8.388	1.301	5.516

Table 5.5: VLog times in seconds for dataset 1 setup 1

run	whole run	loading from File	loading into VLog	solving	querying
1	108.480	31.065	60.568	2.888	13.544
2	107.090	32.024	58.100	2.952	13.538
3	105.281	30.380	55.396	2.995	16.077
AVG	106.950	31.156	58.021	2.945	14.386

Table 5.6: VLog times in seconds for dataset 2 setup 1

run	whole run	loading	solving	querying
1	12.495	6.626	1.060	4.606
2	13.118	6.865	1.052	4.990
3	12.818	6.720	1.079	4.807
AVG	12.810	6.737	1.063	4.801

Table 5.7: VLog times in seconds, for dataset 1 setup 2

run	whole run	loading	solving	querying
1	25.051	12.741	2.082	9.762
2	25.884	12.799	2.089	10.529
3	25.883	12.860	2.135	10.422
AVG	25.606	12.800	2.102	10.238

Table 5.8: VLog times in seconds, for dataset 2 setup 2

6 Conclusion

Lastly let's take a look at what I have accomplished in this thesis and about the next possible steps that can be made to improve my approach. My main goal was it to integrate Clingo into Rulewerk which I have managed to do. However, I made two rather unpleasant discoveries in the process of translating the syntax of Rulewerk into the one of Clingo. First of all existential rules need to by skolemized. This is unavoidable, since Clingo can not handle such rules. This is in case of the LUBM not a big problem, since as we have seen in the Evaluation chapter, only 8 of the 96 rule are existential rules. Troublesome was also the fact, that Clingo can not handle IRIs, which is quite a big problem, since IRIs are important for graph representation in the semantic web technologies, such as OWL 2 [MPSP$^+$12] or RDF [PCBBL14]. But in contrast to the existential rules are IRIs part of every rule and every fact, if we take again a look at the LUBM. Probably this problem can be improved/solved, by maybe passing the internal representation of VLog direktly into Clingo, but it needs further investigation. Another problem I encountered is the huge amount of time, that CSV files need to be loaded through the VLog reasoner in order to subsequently be loaded into Rulewerk. Therefore it would be more efficient, if there was a way for a direct loading into Rulewerk. But besides those problems. The main purpose of integrating Clingo into Rulewerk, which is to compare new algorithms or improvements of the given VLog algorithms to Clingo, can still be served perfectly. Because while testing such algorithms the loading times are not important and hence these problems do not affect the comparison. But as we have seen in the evaluation section, the Clingo reasoner was actually way solver, compared to the VLog reasoner. Nevertheless Clingo is actually capable of adding a lot of settings to the grounding and solving process, thus it maybe just needs some adjustment in consequence to be a worthy opponent to Vlog. Therefore I suggest, that this also needs further investigation.

Bibliography

[Bre19] Steffen Breuer. Ba2020resources. https://github.com/StBreuer/BA2020_resources, 2019.

[CDG+19] David Carral, Irina Dragoste, Larry González, Ceriel Jacobs, Markus Krötzsch, and Jacopo Urbani. Vlog: A rule engine for knowledge graphs. In Chiara Ghidini, Olaf Hartig, Maria Maleshkova, Vojtěch Svátek, Isabel F. Cruz, Aidan Hogan, Jie Song, Maxime Lefrançois, and Fabien Gandon, editors, *Proceedings of the 18th International Semantic Web Conference (ISWC'19) Part II*, volume 11779 of *LNCS*. Springer, October 2019.

[Gag19] Sarah Alice Gaggl. Asp 1. https://iccl.inf.tu-dresden.de/w/images/b/b8/PSSAI2019_L6.pdf, 2019.

[GKKS14] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Clingo = ASP + control: Preliminary report. *CoRR*, abs/1405.3694, 2014.

[GPH05] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. Lubm: a benchmark for owl knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3:158–182, 10 2005.

[Krö18] Markus Krötzsch. The chase. https://iccl.inf.tu-dresden.de/w/images/c/cc/DBT2018-Lecture-19-overlay.pdf, 2018.

[Krö19] Markus Krötzsch. Rules for querying graphs. https://github.com/knowsys/rulewerk/wiki/Rule-syntax-grammar, 2019.

[KSW+19] Roland Kaminski, Torsten Schaub, Philipp Wanko, Javier Romero, and bartbog. Potassco guide version 2.2.0. https://github.com/potassco/guide/releases/tag/v2.2.0, 2019.

[MG20] Maximilian Marx and Larry González. Rule syntax grammar.

https://github.com/knowsys/rulewerk/wiki/Rule-syntax-grammar, 2020.

[MPSP+12] Boris Motik, Peter F. Patel-Schneider, Bijan Parsia, Conrad Bock, Achille Fokoue, Peter Haase, Rinke Hoekstra, Ian Horrocks, Alan Ruttenberg, Uli Sattler, and Michael Smith. Owl 2 web ontology language structural specification and functional-style syntax (second edition).
https://www.w3.org/TR/owl2-syntax/#Ontology_IRI_and_Version_IRI, 2012.

[PCBBL14] Eric Prud'hommeaux, Gavin Carothers, David Beckett, and Tim Berners-Lee. Rdf 1.1 turtle.
https://www.w3.org/TR/turtle/, 2014.

[PS08] Eric Prud'hommeaux and Andy Seaborne. Sparql query language for rdf.
https://www.w3.org/TR/rdf-sparql-query/, 2008.

Copyright Information

If the author used ressources from third parties (texts, images, code) he or she should state the consents of the copyright owners here or cite the given general conditions (e.g. CC/(L)GPL/BSD copyright notices)