

Type-safe Programming for the Semantic Web

by
Martin Gerhard Leinberger

Approved Dissertation thesis for the partial fulfillment of the requirements for a
Doctor of Natural Sciences (Dr. rer. nat.)
Fachbereich 4: Informatik
Universität Koblenz-Landau

Chair of PhD Board: Prof. Dr. Jan Jürjens

Chair of PhD Commission: Prof. Dr. Matthias Gouthier

Examiner and Supervisor: Prof. Dr. Ralf Lämmel, Prof. Dr. Steffen Staab

Further Examiner: Prof. Dr. Ulrike Sattler, Prof. Dr. Viorica Sofronie-Stokkermans

Date of the doctoral viva: 18. December 2020

This thesis will be published in the book series *Studies on the Semantic Web** at IOS Press.

*<http://www.semantic-web-studies.net/>

Abstract

Graph-based data formats are flexible in representing data. In particular semantic data models, where the schema is part of the data, gained traction and commercial success in recent years. Semantic data models are also the basis for the Semantic Web—a Web of data governed by open standards in which computer programs can freely access the provided data. This thesis is concerned with the correctness of programs that access semantic data. While the flexibility of semantic data models is one of their biggest strengths, it can easily lead to programmers accidentally not accounting for unintuitive edge cases. Often, such exceptions surface during program execution as run-time errors or unintended side-effects. Depending on the exact condition, a program may run for a long time before the error occurs and the program crashes.

This thesis defines type systems that can detect and avoid such run-time errors based on schema languages available for the Semantic Web. In particular, this thesis uses the Web Ontology Language (OWL) and its theoretic underpinnings, i. e., description logics, as well as the Shapes Constraint Language (SHACL) to define type systems that provide type-safe data access to semantic data graphs. Providing a safe type system is an established methodology for proving the absence of run-time errors in programs without requiring execution. Both schema languages are based on possible world semantics but differ in the treatment of incomplete knowledge. While OWL allows for modelling incomplete knowledge through an open-world semantics, SHACL relies on a fixed domain and closed-world semantics. We provide the formal underpinnings for type systems based on each of the two schema languages. In particular, we base our notion of types on sets of values which allows us to specify a subtype relation based on subset semantics. In case of description logics, subsumption is a routine problem. For the type system based on SHACL, we are able to translate it into a description logic subsumption problem.

Zusammenfassung

Eine Stärke von graphbasierten Datenformaten ist die Flexibilität bei der Datenmodellierung. Insbesondere semantischen Datenmodellen, bei denen das Schema Teil der Daten ist, wurde in den letzten Jahren viel Aufmerksamkeit geschenkt. Zu den bekanntesten Anwendungsbeispielen von semantischen Datenmodellen gehören die Wissensgraphen von Google und Microsoft. Semantische Datenmodelle liefern auch die Grundlage für das Semantic Web – ein Web der Daten das mit offenen Standards gebaut ist und so Wissen für Programme bereitstellt. Diese Arbeit befasst sich mit der Korrektheit solcher Programme. Während die Flexibilität von solchen semantischen Datenmodellen ihre größte Stärke ist, ist sie auch die größte Fehlerquelle. Ein Programmierer kann leicht einen unintuitiven Randfall übersehen. Solche unbehandelten Randfälle führen oft zu Laufzeitfehlern oder anderen ungewünschten Seiteneffekten. Dabei muss der Fehler nicht sofort auftreten. Je nach Fehler kann ein Program lange korrekt laufen bis der Fehler schlussendlich zum Absturz des Programs führt.

Das Ziel dieser Arbeit ist es solche Fehler, mithilfe von Typsystemen sowie den in Semantic Web üblichen Schemasprachen, zu vermeiden. Insbesondere stützt sich diese Arbeit auf die Ontology Web Language (OWL) und deren theoretische Grundlagen in Form von Beschreibungslogiken sowie der Shape Constraint Language (SHACL), um Typsysteme zu definieren, die getypten Datenzugriff erlauben und dabei typsicher sind. Solche Typsysteme sind eine bewährte Methode um Laufzeitfehler in Programmen zu vermeiden ohne diese auszuführen. Die Semantik beider Schemasprachen basiert auf möglichen Welten, unterscheidet sich aber bei unvollständigem Wissen. Während OWL es erlaubt unvollständiges Wissen durch eine “open-world assumption” zu modellieren benutzt SHACL eine “closed-world assumption”. Diese Arbeit stellt Typsysteme für beide Schemasprachen bereit. Wir interpretieren Typen als Mengen von Werten und definieren Spezialisierung zwischen solchen Typen als Teilmengenbeziehung. Für Beschreibungslogik ist subsumption zwischen Mengen ein Standardproblem. Im Fall des auf SHACL basierten Typsystems zeigen wir das subsumption in ein Beschreibungslogik-Problem überführt werden kann und so gelöst werden kann.

Acknowledgments

This dissertation would not have been possible without the help of my family, friends, and colleagues.

First of all, I want to thank my supervisors, Steffen Staab and Ralf Lämmel, for giving me the opportunity to obtain a Ph.D., their continued support, scientific guidance, and valuable discussions. Furthermore, I am grateful for the advice and feedback of Thomas Gottron, Claudia Schon, Tjitze Rienstra and Matthias Thimm.

I also want to thank my colleagues at WeST for their friendship, feedback, and discussions on all matters. In particular, I want to thank Alex Baier, Daniel Janke, Korok Sengupta, Lukas Schmelzeisen, Philipp Seifer, and Raphael Menges for the important discussions and feedback.

Lastly, I want to thank my family—my parents, my wife Azadeh and my son Kian—for their support and encouragement.

Contents

1. Introduction	1
1.1. Research Questions	3
1.2. Research Contributions	5
1.3. Supporting Publications	6
2. Preliminaries	9
2.1. Type-safe Programming	9
2.1.1. Syntax and Semantics	9
2.1.2. Type System	12
2.1.3. Type Safety	14
2.2. Semantic Web	16
2.2.1. Resource Description Framework	17
2.2.2. SPARQL Conjunctive Queries	19
2.2.3. Description Logics	21
2.2.4. Shape Constraint Language	27
3. A Basic Programming Language (λ-calculus)	35
3.1. The Simply Typed λ -calculus with Subtyping	35
3.1.1. Syntax and Semantics	35
3.1.2. Type System	40
3.1.3. Subtyping	42
3.2. Extensions to the language	45
3.2.1. Recursion	46
3.2.2. Records	48
3.2.3. Lists	50
4. Type Checking with Description Logics	55
4.1. Key Design Principles and Example Use Case	55
4.2. Types for Conjunctive Queries	58
4.3. Core Language	59
4.4. Typecase	63

4.5. Type Safety	67
4.5.1. Soundness of Query Typing	67
4.5.2. Soundness of the Type System	68
4.6. Summary and Discussion	72
5. Type Checking with SHACL	73
5.1. Design Principles and Example Use Case	73
5.2. Types for Conjunctive Queries	76
5.3. Core Language	77
5.4. Type Elaboration	81
5.5. Type Safety	84
5.5.1. Soundness of Query Typing	84
5.5.2. Soundness of the Type System	87
5.6. Summary and Discussion	89
6. Shape Containment	91
6.1. Problem Description	91
6.2. From SHACL to Description Logic	93
6.3. Deciding Shape Containment using Standard Entailment	102
6.4. Effects on Algorithmic Type Checking for λ_{SHACL}	104
6.5. Summary and Discussion	105
7. Related Work	107
7.1. RDF Schema Languages	107
7.2. Containment Problems	108
7.3. Language Integration	108
8. Conclusion	111
Bibliography	115
A. Soundness of λ_{Full}	123
List of Figures	129
List of Algorithms	131
Curriculum Vitae	132

Introduction

Graph-based data models allow for a flexible representation of data that is useful for capturing knowledge. In particular, data based on semantic data models, where conceptualizations and schemata are stored inside the data as part of the graph, have grown considerably and fuel many different applications. Most prominently, the knowledge graphs by Google and Microsoft enhance Internet search. Another example is Wikidata [91], an open-source knowledge graph storing several billion semantic statements which are, among others, used in Wikipedia. Schema.org¹ provides schematic information for data which is being used both to enhance search as well as in personal assistants such as Google Now and Cortana. Based on Schema.org, Google stores more than three trillion facts that are extracted from the Web [73]. A key factor in the popularity of knowledge graphs is their ability to deal with a large variety in the captured knowledge that arises, for example, when integrating different data sources. Figure 1.1 shows an abstracted version of such a knowledge graph.

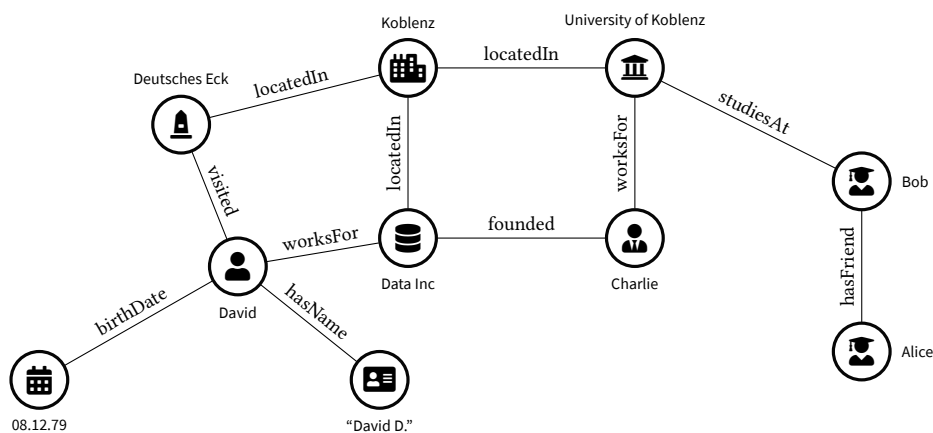


Figure 1.1.: Stylized depiction of a knowledge graph.

Semantic data is also a foundational component in the idea of the *Semantic Web*: A Web of data in which Web sites provide their knowledge in a machine-readable format using open standards².

¹<https://schema.org/>

²<https://www.w3.org/standards/semanticweb/>

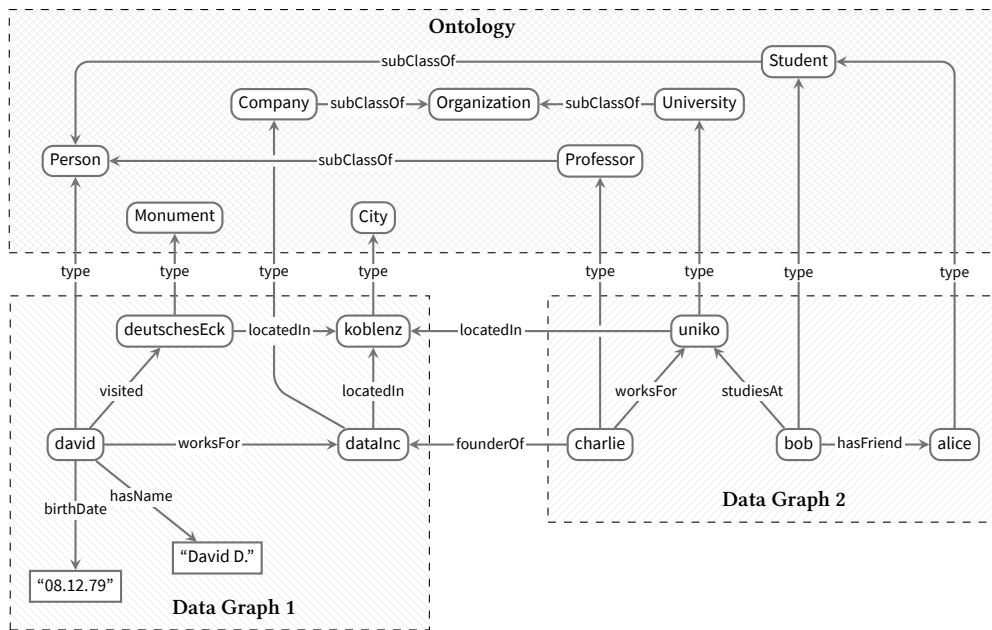


Figure 1.2.: Example that combines the data graphs of two Web sites with an ontology (IRIs abbreviated).

The goal of the Semantic Web is to enable computers to integrate and process knowledge provided through the Web without human intervention [21]. An example for an open standard used in the Semantic Web is RDF [37]. RDF is used to represent graphs in the Semantic Web. Nodes and edges in such graphs are represented by IRIs, essentially allowing for linking graph nodes across different graphs. Another open standard used in the Semantic Web are OWL ontologies [47] which can act as schemas for graphs by introducing types called concept expressions. OWL ontologies conceptualize domains through these types and provide semantics for terms across different Web sites.

As an example, Figure 1.2 depicts a concrete version of Figure 1.1 in which two RDF data graphs about a city and a university are combined using a common ontology to create a larger dataset.³ The first data graph contains a node `david` that represent a `Person` called `"David D."` who visited `deutschesEck`, an instance of `Monument`. Furthermore, he works for the `Company` called `dataInc`. Both, `deutschesEck` und `dataInc` are located in `koblenz`, an instance of `City`. The second data graph contains the node `uniko`, an instance of `University`. `charlie`, an instance of `Professor`, works for `uniko`. `bob`, an instance of `Student`, studies at `uniko` and is friends with `alice`, another instance of `Student`. The two datasets are connected via the relations `-locatedIn-` between `uniko` and `koblenz` as well as `-founderOf-` and through the ontology. The ontology provides terms such as `Person` or `Professor` that are concepts (or types) for the data. Furthermore, it defines the relation between them, for example, that `Student` is subclass of `Person`.

³In general, we avoid writing full IRIs. In graphs, we instead use single terms to depict graph nodes. Furthermore, concepts are capitalized whereas instances use camel case.

While the approach used in the Semantic Web makes data easily accessible and machine-readable, its flexibility incurs costs. Typically, a data graph relies on many different concepts. A programmer must keep track of them including the complex relations between them. This can lead to potentially unintuitive edge cases. For example, a programmer may consider `(david)` as an exemplary instance of `(Person)` and subsequently write a program that queries for all instances of `(Person)` in the data graphs of Figure 1.2 and then accesses the `-hasName→` relation to print the names:

```
1 for x in (query SELECT ?x WHERE {?x type Person})
2   print x.hasName
```

The query will return the nodes `(alice)`, `(bob)`, `(charlie)`, and `(david)`. However, the `-hasName→` relation is only defined for the node `(david)`. In order to cover all edge cases, a programmer must not only specify what happens if no name is known, but also what happens if multiple names are known. Such a missed case is typically only surfacing as a run-time error which causes a program to crash. Typically, programming languages prevent such kind of errors through type systems which prove the absence of run-time errors. For this, they rely on types that approximate the run-time behavior of the program. For example, a program may introduce a variable `person` that is typed with `(Person)`. Subsequently, only expressions that evaluate to an instance of a subtype of `(Person)` can be assigned to this variable—e. g., the first result of a query that selects all instances of `(Student)`.

```
1 var person : Person = head (query SELECT ?x WHERE { ?x type Student })
```

However, programming languages are not aware of this form of types. This means that types already used in the Semantic Web and their relations are not considered during type checking. In this thesis, we aim to bridge the gap between types in the Semantic Web and types in the programming language.

1.1. Research Questions

The main goal of the thesis is to investigate *type systems based on schematic descriptions for RDF data graphs*. Type systems are an established method in the field of software engineering that allow for proving the absence of run-time errors without executing the program (c. f. [81]). In essence, they work by classifying the syntactic elements of the program according to the values they compute. Using such a classification, a type system that is sound can guarantee the absence of certain run-time errors. We focus on schematic descriptions based on OWL [47] as well as schematic descriptions based on the Shape Constraint Language (SHACL) [59]. While both schema languages use possible world semantics, they serve different purposes.

OWL, rooted in description logics, comprise logical axioms. For example, the axiom description logic `Person ⊑ ∃ hasName.⊤`, built with the two concept expressions `Person` and `∃ hasName.⊤`, defines that all instances of `Person` must have a name. However, OWL ontologies also allow for mod-

eling incomplete knowledge. Even though `alice`, `bob` and `charlie` have no name, the graph as shown in Figure 1.2 is still valid with respect to the axiom—their names simply constitute incomplete knowledge. Furthermore, OWL uses an open-world semantics. While `bob` is known to be an instance of `Student`, it is unknown whether he is an instance of `Professor` rather than false. This thesis aims to use concept expressions like `Person` and $\exists \text{hasName}.\top$ as types representing sets of values. This new form of types is then used to type terms of a programming language extended to support querying of graph data.

SHACL shape definitions, on the other hand, provide integrity constraints for RDF graphs. Similar to XML Schema or JSON Schema, they allow for validating RDF graphs—that is, deciding whether the RDF graph conforms to the given SHACL shape definitions. For example, a shape definition requiring every instance of `Person` to have at least one `-hasName→` relation is shown in Figure 1.3. In this case, the

```
1 :PersonShape a sh:NodeShape;
2   sh:targetClass :Person;
3   sh:property [
4     sh:path :hasName;
5     sh:minCount 1;
6   ].
```

Figure 1.3.: SHACL shape that enforces that every instance of `Person` has a `-hasName→` relation.

graph shown in Figure 1.2 is not valid with respect to the SHACL shape `PersonShape`. The thesis aims to use shape names such as `PersonShape` as types which represent graph nodes. These types are then used to type terms of a programming language featuring embedded querying.

Since we use the notion that types of a programming language represent sets of values, our subtype relation is based on subset semantics. In case of description logics, subsumption of concept expressions is a basic reasoning task [19]. However, in case of SHACL shapes, the problem of shape containment—that is, deciding whether two shapes representing sets of graph nodes are in a subset relation—is an open problem that is addressed in this thesis. Contrary to the integration of other data models, the complexity of the subtype relation makes a mapping to types in a programming language (c. f. [75] for SQL) not suitable due to complexity of the subtype relation.

In summary, the following research questions will be addressed:

Research Question 1: How can OWL ontologies be leveraged to achieve a type-safe programming language for working with RDF graphs?

We propose to use OWL ontologies for type checking program code. We leverage description logics, the theoretic foundation of OWL ontologies, and use concept expressions as types. We

then require definitions for typing program expressions and queries. In particular, the type system must consider the open-world semantics employed by OWL which allows for modelling incomplete knowledge.

Research Question 2: How can SHACL be leveraged to achieve a type-safe programming language for working with RDF graphs?

The open-world assumption employed by description logics may be counterintuitive for programmers as type systems typically rely on a closed world. We therefore also investigate a type system based on an abstraction of the Shape Constraint Language (SHACL) in which so called shape expressions constitute types. Again, this requires definitions for the typing of program expressions as well as subtyping between shapes.

Research Question 3: How can containment of SHACL shape expressions be decided?

We base our subtyping relation between types on subset semantics. In case of description logics, subsumption is a basic reasoning task. However, subsumption or containment between SHACL shapes is not used in the validation of RDF graphs. Therefore, the problem of shape containment has not been considered so far. In the context of type checking however, it is required for deciding subtyping. We therefore investigate how shape containment can be decided.

1.2. Research Contributions

To address the research questions described in the previous section, we define two languages λ_{DL} and λ_{SHACL} and their associated type systems. Both languages allow for defining programs that are written with respect to an RDF data graph and either an OWL ontology or SHACL shape definitions. We identify suitable types that represent RDF graph nodes based on the notion that types represent sets of values. We define operations for working with RDF graphs—in particular, querying as well as traversing the graph. We then continue to show the type safety of the defined languages.

Chapter 2 recaps the basics of type systems and the Semantic Web, in particular, RDF, SPARQL, description logics and SHACL. Chapter 3 then introduces the λ -calculus which acts as a basic programming language. Both chapters build upon existing work. In particular, Chapter 3 uses definitions from *Types and Programming Languages* [81] whereas Chapter 2 builds upon various sources. The language λ_{DL} is then described in Chapter 4 and addresses the first research question. Using DL concept expressions as types, it defines how concept expressions are derived from queries and how they are assigned to program expressions. The language λ_{SHACL} , described in Chapter 5, addresses the second research question. It relies on an abstraction of SHACL shape expressions as types. Lastly, we address the third research question in Chapter 6. We investigate the correspondence between shape containment and subsumption in DL concept expressions. In particular, we show that for a subset of the SHACL dialect used in this thesis, deciding shape containment through a translation into a DL concept subsumption

problem is sound and complete. For the complete SHACL dialect as used in this thesis, we show that our translation into a description logic problem is a sound but incomplete approach.

In short, the research contributions of this thesis are:

Contribution 1: We define a type system based on description logics, the foundation of OWL ontologies.

We then show that the type system of the language λ_{DL} is sound (also known as type safety).

Contribution 2: We define a type system based on a logical abstraction of SHACL shape expressions.

We then show that the type system of the resulting language λ_{SHACL} is sound.

Contribution 3: For SHACL shape containment, we provide a transformation from a SHACL shape containment problem to a concept subsumption problem in description logic. We show that for a subset of the SHACL definitions used in this thesis, the translation is sound and complete. For the complete definitions of SHACL as used in this thesis, deciding shape containment through DL concept subsumption is sound but incomplete.

1.3. Supporting Publications

This thesis is supported by several publications, listed in reverse-chronological order:

- (1) Leinberger, M., Seifer, P., Rienstra, T., Lämmel, R., Staab, S.: Deciding SHACL Shape Containment Through Description Logics Reasoning. In: *Proceedings of the 19th International Semantic Web Conference (ISWC 2020)*. LNCS, vol. 12506, pp. 366–383. Springer (2020).
- (2) Leinberger, M., Seifer, P., Schon, C., Lämmel, R., Staab, S.: Type Checking Program Code Using SHACL. In: *Proceedings of the 18th International Semantic Web Conference (ISWC 2019)*. LNCS, vol. 11778, pp. 399–417. Springer (2019).
- (3) Seifer, P., Leinberger, M., Lämmel, R., Staab, S.: Semantic Query Integration With Reason. *Programming Journal* 3(3), 13 (2019).
- (4) Hartenfels, C., Leinberger, M., Lämmel, R., Staab, S.: Type-Safe Programming with OWL in Semantics4J. In: *Proceedings of the ISWC 2017 Posters & Demonstrations and Industry Tracks, 16th International Semantic Web Conference (ISWC 2017)*. CEUR Workshop Proceedings, vol. 1963.
- (5) Leinberger, M., Lämmel, R., Staab, S.: The Essence of Functional Programming on Semantic Data. In: *Proceedings of the 26th European Symposium on Programming (ESOP 2017)*. LNCS, vol. 10201, pp. 750–776. Springer (2017).
- (6) Leinberger, M., Scheglmann, S., Lämmel, R., Staab, S., Thimm, M., Viegas, E.: Semantic Web Application Development with LITEQ. In: *Proceedings of the 13th International Semantic Web Conference (ISWC 2014)*. LNCS, vol. 8797, pp. 212–227. Springer (2014).

- (7) Scheglmann, S., Leinberger, M., Lämmel, R., Staab, S., Thimm, M.: Property-based typing with LITEQ. In: *Proceedings of the 13th International Semantic Web Conference (ISWC 2014), Posters & Demonstrations Track*. CEUR Workshop Proceedings, vol. 1272, pp. 149–152. CEUR-WS.org (2014)
- (8) Scheglmann, S., Lämmel, R., Leinberger, M., Staab, S., Thimm, M., Viegas, E.: IDE Integrated RDF Exploration, Access and RDF-Based Code Typing with LITEQ. In: *The Semantic Web: ESWC 2014 Satellite Events (ESWC 2014)*. LNCS, vol. 8798, pp. 505–510. Springer (2014).

Publication (5) contains the definitions for λ_{DL} and acts as the foundation for Chapter 4. Publication (3) describes an implementation of λ_{DL} leveraging compiler extensions in the programming language Scala. Chapter 4 uses some of the theoretical parts that are part of this publication. Another implementation of λ_{DL} based on an extended compiler for the language Java is described in (4). The language λ_{SHACL} is described in publication (2) which serves as the basis for Chapter 5. Lastly, Chapter 6 is based on publication (1). Other publications include (6) which describes a mapping-based approach for programming with RDF graphs and ontologies. The publication influenced this thesis as the defined type systems solve many of the problems which limited the usefulness of (6). Publications (7) and (8) describe other mapping-based approaches to programming with RDF.

Preliminaries

In order to investigate type-safe programming for RDF data graphs, we recap the basics of type systems as well as the components of the Semantic Web.

2.1. Type-safe Programming

While modern software engineering and programming language theory knows many formal methods, *type systems* are among the most well-known. According to Pierce [81]

A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kind of values they compute.

In this thesis, we rely on type systems to show the absence of *run-time errors*. That is, errors that occur during evaluation of a program and cause it to abort execution. While some run-time errors are obvious, others may only occur after a program ran for a long time. We use type systems that rely on a static analysis of a program—that is, the syntactic elements of the program are analyzed without actually evaluating the program—to show that the program will not abort due to a run-time error. A type system that can guarantee the absence of run-time errors is called *type safe*. As Milner [70] puts it, a well-typed program cannot “go wrong”. However, we will highlight what type safety means through a small programming language named \mathbb{NB} that features Booleans and numerical expressions. The definitions used in this section are mainly taken from the textbook *Types and Programming Languages* [81], albeit sometimes slightly altered.

2.1.1. Syntax and Semantics

Syntax Programming languages allow for writing programs—phrases that represent computations. Following the definitions of Pierce [81], we use the word *term* (abbreviated as t) to represent such computations whereas we use the word *expression* for all sorts of syntactic constructs. We highlight expressions when they occur in running text for readability. The language \mathbb{NB} which we use as an example contains only a few terms. Namely, it contains Boolean constants `true` and `false`. It also contains if-then-else expressions of the form `if t_1 then t_2 else t_3` , where t_1 constitutes the guard and

\mathbb{NB} (*untyped*)

Syntax		Evaluation	$t \longrightarrow t'$
$t ::=$	<i>terms:</i>	if true then t_2 else t_3 $\longrightarrow t_2$	(E-IFTRUE)
true	constant true	if false then t_2 else t_3 $\longrightarrow t_3$	(E-IFFALSE)
false	constant false	$\frac{t_1 \longrightarrow t'_1}{\mathbf{if } t_1 \mathbf{ then } t_2 \mathbf{ else } t_3 \longrightarrow \mathbf{if } t'_1 \mathbf{ then } t_2 \mathbf{ else } t_3}$	(E-IF)
if t then t else t	if-then-else	$\frac{t_1 \longrightarrow t'_1}{\mathbf{succ } t_1 \longrightarrow \mathbf{succ } t'_1}$	(E-SUCC)
0	constant zero	$\mathbf{pred } 0 \longrightarrow 0$	(E-PREDZERO)
succ t	successor	$\mathbf{pred } (\mathbf{succ } nv_1) \longrightarrow nv_1$	(E-PREDSUCC)
pred t	predecessor	$\frac{t_1 \longrightarrow t'_1}{\mathbf{pred } t_1 \longrightarrow \mathbf{pred } t'_1}$	(E-PRED)
iszero t	zero test	$\mathbf{iszero } 0 \longrightarrow \mathbf{true}$	(E-ISZEROZERO)
$v ::=$	<i>values:</i>	$\mathbf{iszero } (\mathbf{succ } nv_1) \longrightarrow \mathbf{false}$	(E-ISZEROSUCC)
true	true value	$\frac{t_1 \longrightarrow t'_1}{\mathbf{iszero } t_1 \longrightarrow \mathbf{iszero } t'_1}$	(E-ISZERO)
false	false value		
nv	numeric value		
$nv ::=$	<i>numeric values:</i>		
0	zero value		
succ nv	successor value		

Figure 2.1.: Syntax and evaluation rules for arithmetic expressions (\mathbb{NB}).

t_2 and t_3 the different branches of the expression. Furthermore, the language contains the numeric constant `0`, the arithmetic operators `succ` and `pred` for successor and predecessor¹ as well as a predicate `iszero` that evaluates to `true` when applied to the constant `0` and `false` when applied to some other number. Figure 2.1 summarizes the grammar of the language (left-hand column). The language is *untyped*, meaning that there are no syntactic elements or rules for types yet. To improve readability, we sometimes add parentheses to the phrases of the language even though they are not explicitly mentioned in the grammar.

Any program of the language \mathbb{NB} is just a term built from the grammar given in Figure 2.1. As an example, consider the program `iszero (succ 0)`. Intuitively, this program represents a computation that evaluates to `false`. A subset of terms called *values* (v) constitute possible final results of the

¹To simplify expressions, we sometimes use decimal numbers instead of the actual syntactic representations. For example, we use `1` and `2` to represent `succ 0` and `succ (succ 0)`.

evaluation of a term—such as `true`, `false`, or numeric values `nv`.

Semantics In terms of semantics of programming languages, we rely on *small step operational semantics*. That is, we define the behavior of the language through an *evaluation relation*, sometimes also called *reduction relation*, $t \longrightarrow t'$ expressing that a term t can be reduced to t' in one step. The evaluation relation is defined through a set of inference axioms (right-hand column of Figure 2.1). Furthermore, we use $t \longrightarrow^* t'$ to denote that a term t is reduced to t' by repeated application of the evaluation rules. In essence, the evaluation relation defines an abstract machine that interprets the program. The *meaning* of a term t is the final state that the machine reaches [81]. The meaning of term t is therefore the term t' that is produced by repeatedly applying the evaluation rules $t \longrightarrow^* t'$ such that t' cannot be reduced any further.

Consider the rule E-IFTRUE. The rule says that if the term is an if-then-else expression where the guard is the constant `true`, then the term t can be reduced to the then-part of the expression, namely t_2 . Likewise, rule E-IF says that if the guard t_1 of the if-then-else expression can take a step and evaluate to t'_1 , then the whole expression `if t_1 then t_2 else t_3` evaluates to `if t'_1 then t_2 else t_3` . Consider the following statement:

$$\text{if iszero 0 then true else false} \longrightarrow \text{if true then true else false}$$

The derivability of this statement—that is, it is possible to reduce t to t' by applying the reduction rules—is shown in the following derivation tree:

$$\frac{\frac{}{\text{iszero 0} \longrightarrow \text{true}} \text{E-ISZEROZERO}}{\text{if iszero 0 then true else false} \longrightarrow \text{if true then true else false}} \text{E-IF}$$

Likewise, repeated application of the evaluation rules looks as follows:

$$\text{if iszero 0 then true else false} \longrightarrow^* \text{true}$$

The term `true` cannot be evaluated any further since no rules apply anymore. If no evaluation rule applies to a term anymore, then the term is said to be in *normal form*. Ultimately, terms are expected to be reduced to values. In the example above, the term is reduced to the value `true`. By definition, values are in normal form. However, some terms cannot be reduced further even though they are not values. In case of the term `iszero true`, no evaluation rule applies. The term is in normal form, but not a value. Such a term is *stuck*. Intuitively, an abstract machine or interpreter for a language would not know what to do with such a term. Execution of the program has “gone wrong” and the term is meaningless. Subsequently, the interpreter will raise an error and abort execution. We use “stuckness” as a simple notion of a run-time error.

Definition 1 (Run-time error). *A term t is in normal form if no evaluation rule applies to it—i. e., there is no t' such that $t \rightarrow t'$. If a term t is in normal form but not a value, then we say that t is stuck. A stuck term represents a run-time error.*

2.1.2. Type System

A term either evaluates to a value or it gets stuck at some point. A stuck term represents a meaningless program or run-time error. Assigning types to terms allows for proving that a term will definitely not get stuck without actually evaluating the term. One common view on the semantics of types, which is also adopted in this thesis, is to see a type T as a set of values [31]. A term t having a type T then means that it is possible to statically show—that is, without actually evaluating t —that t will evaluate to a value belonging to T .

Types are assigned to terms through a set of inference rules. This set of inference rules define the typing relation $t : T$. Figure 2.2 summarizes the newly introduced syntactic forms (types) and inference rules. Rules T-TRUE and T-FALSE assign the type `Bool` to the values `true` and `false`. Rule T-IF assigns a type T to the if-then-else expression depending on the type of its subexpressions. If the guard t_1 is assigned the type `Bool` and both t_2 and t_3 are assigned the type T , then the complete expression is assigned to the type T . Likewise, the value `0` is of type `Nat` (rule T-ZERO). For a successor expression `succ t_1` , if t_1 is typed with `Nat`, then the expression also has type `Nat` (rule T-SUCC). Rule T-PRED works similarly. If t_1 in `pred t_1` is of type `Nat`, then the complete expression is also assigned to the type `Nat`. In case of the term `iszero t_1` , rule T-ISZERO assigns `Bool` in case that the subexpression t_1 is typed with `Nat`. As with the evaluation relation, consider the following statement as a simple example of the typing relation:

`if iszero 0 then true else false : Bool`

The following derivation tree shows how the type `Bool` can be derived for the term:

$$\frac{\frac{\frac{}{0 : \text{Nat}}{\text{iszero } 0 : \text{Bool}} \text{T-ISZERO} \quad \frac{}{\text{true} : \text{Bool}} \text{T-TRUE} \quad \frac{}{\text{false} : \text{Bool}} \text{T-FALSE}}{\text{if iszero 0 then true else false} : \text{Bool}} \text{T-IF}}{\text{if iszero 0 then true else false} : \text{Bool}} \text{T-IF}$$

A term t is called *well-typed* if it is possible to assign a type. Contrary to that, some terms cannot be assigned a type. For example, for the term `iszero true`, no rule exists such that a type can be assigned. The general idea of a type system is that a type is only assigned to a term if it can be shown that the term will not get stuck. The derivation of the type acts as a proof that the term does not “go wrong” when evaluated. Subsequently, only programs that are well-typed should be allowed to be interpreted or “executed” through the evaluation relation.

\mathbb{NB} (<i>typed</i>)		Extends \mathbb{NB} (Figure 2.1)	
<i>New syntactic forms</i>		<i>Typing rules</i> $t : T$	
$T ::=$	<i>types:</i>	$\text{true} : \text{Bool}$	(T-TRUE)
Bool	type of booleans	$\text{false} : \text{Bool}$	(T-FALSE)
$ \text{Nat}$	type of natural numbers	$\frac{t_1 : \text{Bool} \quad t_2 : T \quad t_3 : T}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T}$	(T-IF)
		$0 : \text{Nat}$	(T-ZERO)
		$\frac{t_1 : \text{Nat}}{\text{succ } t_1 : \text{Nat}}$	(T-SUCC)
		$\frac{t_1 : \text{Nat}}{\text{pred } t_1 : \text{Nat}}$	(T-PRED)
		$\frac{t_1 : \text{Nat}}{\text{iszero } t_1 : \text{Bool}}$	(T-ISZERO)

Figure 2.2.: Typing rules for arithmetic expressions (\mathbb{NB}).

Proposition 1 (Well-typed). *A term t is well-typed (also called typeable) if there is some T such that $t : T$. A well-typed term is not stuck—it is either a value or it can take a step through the evaluation relation.*

To reiterate, the derivation of the type of a well-typed program acts as a proof for the absence of errors. A type system therefore gives a guarantee of the absence of run-time errors. Compared to testing a program through evaluation, deriving a type has several advantages. A run-time error may only occur in very specific path through a program. The derivation of a type always considers all possible paths through a program. Evaluation of a program may result in some intended side-effects, such as the modification of data. However, this is undesired when simply testing the program. Lastly, deriving a type through a static analysis of the program is typically faster than executing the program.

However, it is important to notice that being well-typed is a sufficient but not necessary condition for not getting stuck. Or, in other words: Not being well-typed does not automatically mean that a term will get stuck. The term `if true then true else 0` is not typeable as the two branches of the if-then-else expression have different types. Yet, evaluation of the term will never get stuck. Restricting evaluation to only well-typed terms is a trade-off. One is trading expressiveness (as some programs that evaluate perfectly fine are forbidden) for safety.

2.1.3. Type Safety

The most basic property of any type system is *type safety* (also called *soundness*): A well-typed program does not get stuck during evaluation. Showing that a type system is sound proceeds in two steps:

Progress: A well-typed term is not stuck—it is either a value or it can be reduced further.

Preservation: If a well-typed term is reduced by one step, then the result is also well-typed.

Intuitively, if both properties hold for a language, it is impossible for a term to be stuck.

The first step to showing these two properties is the so called *canonical forms lemma*. The lemma notes the forms of well-typed values that can exist in the language (see [81]).

Lemma 1 (Canonical forms of \mathbb{NB}). *Let v be a well-typed value. Then one of the following must be true:*

1. *If v is a value of type \mathbf{Bool} , then either $v = \mathbf{true}$ or $v = \mathbf{false}$.*
2. *If v is a value of type \mathbf{Nat} , then v is a numerical value nv according to the grammar defined in Figure 2.1.*

Given Lemma 1, progress and preservation can now be shown. Progress, saying that a well-typed term is either a value or it can take a step, is a structural induction over the typing relation $t : T$. For each case, evaluation rules that are applicable are listed.

Theorem 1 (Progress). *Let t be a well-typed term—that is, $t : T$ for some T . Then either t is a value or there is some t' with $t \longrightarrow t'$ [81].*

Proof. By induction on the derivation of $t : T$. T-TRUE, T-FALSE and T-ZERO are immediate since t is a value. For the remaining cases, the argument is as follows:

T-IF $t = \mathbf{if } t_1 \mathbf{ then } t_2 \mathbf{ else } t_3$, $t_1 : \mathbf{Bool}$, $t_2 : T$, $t_3 : T$.

By induction hypothesis, either t_1 is a value or it can take a step. If t_1 is a value, then by Lemma 1, it is either \mathbf{true} or \mathbf{false} , in which case either E-IFTRUE or E-IFFALSE apply. If t_1 is not a value but can take a step, then rule E-IF applies.

T-SUCC $t = \mathbf{succ } t_1$, $t_1 : \mathbf{Nat}$.

By induction hypothesis, t_1 is either a value or it can take a step. If it is a value, then by Lemma 1, it is a numerical value, therefore, the complete term is a numerical value (c. f. Figure 2.1). If t_1 can take a step, then rule E-SUCC applies.

T-PRED $t = \mathbf{pred } t_1$, $t_1 : \mathbf{Nat}$.

By induction hypothesis, t_1 is either a value or it can take a step. If it is a value, then by Lemma 1, either rule E-PREDZERO or rule E-PREDSUCC applies. If it can take a step, rule E-PRED applies.

T-ISZERO $t = \text{iszero } t_1, \quad t_1 : \text{Nat}.$

By induction hypothesis, t_1 is either a value or it can take a step. If it is a value, then by Lemma 1, it is a numerical value. Therefore, either rule E-ISZEROZERO or rule E-ISZEROSUCC applies. If it can take a step, then rule E-ISZERO applies.

□

Preservation is saying that if a well-typed term is reduced by one step, then the type is preserved. Again, the proof consists of a structural induction over the typing relation $t : T$. In each case, possibly applicable evaluation rules are listed and the resulting t' then examined with respect to its type.

Theorem 2 (Preservation). *Let t be a well-typed term—that is, $t : T$ for some T . If $t \longrightarrow t'$, then $t' : T$ [81].*

Proof. By induction on the derivation of $t : T$. Again, T-TRUE, T-FALSE and T-ZERO are immediate as they are values. For the remaining cases, the argument is as follows:

T-IF $t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3, \quad t : T, \quad t_1 : \text{Bool}, \quad t_2 : T, \quad t_3 : T.$

There are three rules by which t' can be derived:

E-IFTRUE $t_1 = \text{true}, \quad t' = t_2.$

If $t \longrightarrow t'$ is derived using rule E-IFTRUE, then t_1 must be `true` and the resulting term is t_2 .

By assumption of the T-IF case, as $t_2 : T$, the type is preserved.

E-IFFALSE Analogous to E-IFTRUE.

E-IF $t_1 \longrightarrow t'_1, \quad t' = \text{if } t'_1 \text{ then } t_2 \text{ else } t_3.$

By induction hypothesis, $t_1 \longrightarrow t'_1$ preserves the type. This means that the type for the whole term is preserved as rule T-IF can be applied again, yielding $t' : T$.

T-SUCC $t = \text{succ } t_1, \quad t : \text{Nat}, \quad t_1 : \text{Nat}.$

Only rule E-SUCC can be used to derive t' , in which t_1 takes a step ($t_1 \longrightarrow t'_1$). By induction hypothesis, this preserves the type ($t'_1 : \text{Nat}$). Rule T-SUCC can be applied again, yielding $t' : \text{Nat}$.

T-PRED $t = \text{pred } t_1, \quad t : \text{Nat}, \quad t_1 : \text{Nat}.$

There are three rules by which t' can be derived:

E-PREDZERO $t_1 = 0, \quad t' = 0.$

By rule T-ZERO, $t' : \text{Nat}$. Therefore, the type is preserved.

E-PREDSUCC Similar to E-PREDZERO.

E-PRED $t_1 \longrightarrow t'_1, \quad t' = \text{pred } t'_1.$

By induction hypothesis, $t_1 \longrightarrow t'_1$ preserves the type ($t'_1 : \text{Nat}$). Therefore, rule T-PRED applies again.

T-ISZERO $t = \text{iszero } t_1, \quad t : \text{Bool}, \quad t_1 : \text{Nat}.$

There are three rules by which t' can be derived:

E-ISZEROZERO $t_1 = 0, \quad t' = \text{true}.$

By rule T-TRUE, the type is preserved since $t' : \text{Bool}$.

E-ISZEROSUCC Same as rule E-ISZEROZERO.

E-ISZERO $t_1 \longrightarrow t'_1, \quad t' = \text{iszero } t'_1.$

By induction hypothesis, $t_1 \longrightarrow t'_1$ preserves the type. Since $t'_1 : \text{Nat}$, rule T-ISZERO applies again and the type is preserved since $t' : \text{Bool}$.

□

Given that a well-typed term is either a value or it can be reduced (progress) and that if a well-typed term is reduced, the result is well-typed again, it follows that the type system is sound. A term cannot be stuck if the term is well-typed. As we use stuckness as a notion for a run-time error, a well-typed program does not contain run-time errors.

2.2. Semantic Web

While the modern World Wide Web is huge both in size as well as knowledge stored within it, there is a fundamental flaw when it comes to using this knowledge within intelligent software agents. Its representation language HTML is intended for presentation to humans rather than machines. This makes it difficult for a software agent or application to extract the knowledge. Even more so if knowledge from multiple Web sites must be combined to yield an answer. The goal of the *Semantic Web* is to enable computers to integrate and process knowledge provided through the Web without human intervention [21]. This is achieved through several standards as defined by the W3C of which we use a subset. For one, the *Resource Description Framework* or *RDF* [37] that is used to represent data. Second, the query language *SPARQL* that is used to retrieve information from RDF data [83]. Third, ontologies that establish common terminologies [47], give formally defined meanings to this terminology, and provide rich conceptual schemas [86, 53]. Last, the *Shapes Constraint Language* (*SHACL*) which uses integrity constraints to validate RDF data [59].

This thesis aims to improve the development of software agents or programs that work with data from the Semantic Web. That is, programs access RDF data through SPARQL queries and then proceed to process this data. Ontologies and SHACL provide information about the structure of the data, which is in turn used for proving the absence of run-time errors in the program. Importantly, this thesis is technically not limited to the Semantic Web, but is rather concerned with semantic data. That is, data which is organized such that it can be interpreted without human intervention. However, we base

this work on the Semantic Web technology stack as this provides formal foundations and well-defined behavior.

2.2.1. Resource Description Framework

The *Resource Description Framework (RDF)* [37] is a language for describing entities and their relations [53]. RDF relies on *International Resource Identifiers (IRIs)* for the identification of entities (sometimes called objects) as well as relations. First, this allows for avoiding conflicts in names because IRIs introduce namespaces. Second, it allows for consistently referencing entities or relations that live outside of the current data set. Consider this example sentence:

Bob has a friend called Alice.

This sentence contains the two entities *Bob* and *Alice* as well as the relation *hasFriend*. In RDF, the entity for Bob could for example be represented by the IRI `http://example.org/bob` whereas the *hasFriend* relation could be represented by the IRI `http://example.org/hasFriend`. In this thesis, we typically avoid giving full IRIs and just write `(bob)` to reference the entity. While some relations such as `-hasFriend→` are defined for a specific domain, other relations such as the `-type→` relation², used to indicate that something is an instance of a *concept* or *class*, are universal. For example, `(bob)` may be an instance of the concept `(Student)`.

RDF datasets, also called RDF data graphs, describe labelled directed graphs where entities are the nodes and relations form the edges of the graph. The example sentence used above can be represented using the graph `(bob)-hasFriend→(alice)`. RDF graphs can be represented as sets of triples. Each triple mimics a basic sentence consisting of a *subject*, a *predicate* (or *property*) as well as an *object*. In this thesis, we use the words predicate, property and relation interchangeably. Subject and object represent nodes such as `(bob)` and predicates represent edges such as `-hasFriend→`. The example sentence can therefore also be represented by the triple `(bob, hasFriend, alice)`. Figure 2.3 shows an RDF data graph containing the knowledge that `(bob)` is a `(Student)` and a `(Person)` who has a friend `(alice)` who is also an instance of `(Person)`, both as a graph as well as a set of triples.

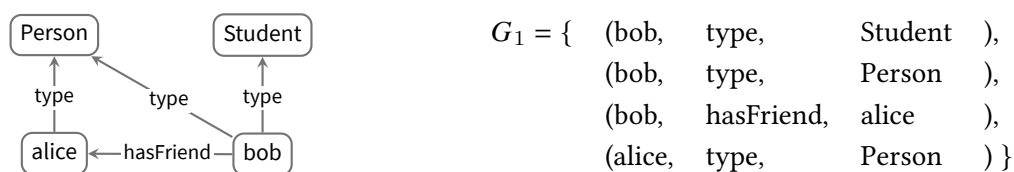


Figure 2.3.: An example of an RDF graph G_1 (left) and its representation as a set of triples (right).

Besides IRIs that identify entities, RDF also features *literals* and *blank nodes*. Literals are used to represent primitive values such as strings, e.g., the name of Bob in the triple `(bob)-hasName→"Bob B."`.

²The full IRI of type is `http://www.w3.org/1999/02/22-rdf-syntax-ns#type`.

Blank nodes may refer to entities or literals but are used to model incomplete information. In that respect, they behave similarly to existentially quantified variables in first-order logic [53]. For example, the triple $(\text{bob})\text{-studiesAt}\rightarrow(\text{b}_1)$, where (b_1) represents a blank node, models the statement that (bob) studies at (b_1) . However, we have no further information about (b_1) .

Definition 2 (RDF Graph). *Let \mathcal{I} be the set of all IRIs, \mathcal{L} the set of all literals and \mathcal{B} the set of all blank nodes. An RDF graph G is a set of triples of the form $(\text{subject}, \text{predicate}, \text{object}) \in (\mathcal{I} \cup \mathcal{B}) \times (\mathcal{I}) \times (\mathcal{I} \cup \mathcal{B} \cup \mathcal{L})$.*

We use G to denote a single RDF graph whereas \mathcal{G} refers to all possible RDF graphs. We use $\text{Nodes}(G) = \{x \mid (x, \text{predicate}, \text{object}) \in G \vee (\text{subject}, \text{predicate}, x) \in G\}$ to refer to the set of nodes of G whereas o refers to an individual node. For example, $\text{Nodes}(G_1) = \{\text{alice}, \text{bob}, \text{Student}, \text{Person}\}$. We sometimes use the expression *object* to refer to an graph node. $\text{Prop}(G) = \{x \mid (o, x, o') \in G\}$ denotes the set of properties with p denoting an individual property. Lastly, we also use $p(G) = \{(x, x') \mid (x, p, x') \in G\}$ to denote the set of all pairs (o, o') such that there is a connection of o to o' via p in graph G . Furthermore, we use $\text{Concepts}(G) = \{x \mid (o, \text{type}, x) \in G\}$ to denote the set of concepts, e. g. $\text{Concepts}(G_1) = \{\text{Student}, \text{Student}\}$, that are used in the RDF graph.

```

1  :bob
2   rdf:type :Student;
3   rdf:type :Person;
4   :hasFriend :alice.
5
6  :alice
7   rdf:type :Person.
```

Figure 2.4.: RDF graph G_1 serialized using N3 Notation (namespace definitions are omitted).

When giving examples, we typically either draw the graph or provide the triples of the graph. However, when many syntactic elements are involved, e. g., when giving examples for SHACL shapes (see Section 2.2.4), we resort to serialized graphs. Figure 2.4 depicts graph G_1 serialized in N3 Notation. N3 Notation features a normal triple structure where a “.” ends a triple (line 6–7). A shortcut is the “;” symbol which allows for omitting repeated subjects (line 1–4). To reiterate, RDF graph nodes are represented through IRIs. We abbreviate IRIs in serialized RDF graphs using namespaces. We use a namespace without prefix to represent graph nodes coming from our example domain (e. g., :bob). Relations that include a namespace, such as `rdf:type` are indicate globally defined relations by the W3C.

2.2.2. SPARQL Conjunctive Queries

RDF data graphs can be queried via the SPARQL standard [83]. For this, RDF data graphs are typically stored in special databases, so called triplestores, that provide efficient implementations of the SPARQL query evaluation rules. However, in this thesis, we abstract away from them and only mention them in passing whenever appropriate. We instead simply define query evaluation over an RDF graph.

SPARQL, at its heart, is a *pattern matching* language. We focus on a core fragment of SPARQL called *conjunctive queries (CQs)*. That is, our queries are conjunctions of triple patterns that use variables only in place of graph nodes, not in place of properties [24]. This constitutes a widely used subset of SPARQL queries [80]. Figure 2.5 summarizes syntax and evaluation rules of the query language.

Conjunctive Queries (CQs)

Syntax		Evaluation	$\llbracket q \rrbracket_G$
$q ::= \bar{x} \leftarrow gp$	query	$\llbracket x r o \rrbracket_G = \{\mu \mid (\mu(x), o) \in r(G)\}$	(Q-SVAR)
$gp ::=$	graph pattern:	$\llbracket o r x \rrbracket_G = \{\mu \mid (o, \mu(x)) \in r(G)\}$	(Q-OVAR)
$gp \wedge gp$	conjunction	$\llbracket x_1 r x_2 \rrbracket_G = \{\mu \mid (\mu(x_1), \mu(x_2)) \in r(G)\}$	(Q-VARS)
$ (x r o)$	subject var pattern	$\llbracket gp_1 \wedge gp_2 \rrbracket_G = \llbracket gp_1 \rrbracket_G \bowtie \llbracket gp_2 \rrbracket_G$	(Q-CONJ)
$ (o r x)$	object var pattern	where $\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1,$	
$ (x r x)$	subject object var pattern	$\mu_2 \in \Omega_2 \text{ are compatible mappings}\}$	
$r ::=$	path expression:	$\llbracket \bar{x} \leftarrow gp \rrbracket_G = \{\mu _{\bar{x}} \mid \mu \in \llbracket gp \rrbracket_G\}$	(Q-PROJ)
p	(property)		
$ r^-$	(inverse path)		

Figure 2.5.: Syntax and Evaluation rules of conjunctive queries (CQs).

Syntax We use the metavariable x to denote a variable and use \bar{x} to denote a sequence of variables x_1, \dots, x_n . Furthermore, we use o to denote an RDF graph node. Syntax is then summarized in Figure 2.5 (left side). A query $q = \bar{x} \leftarrow gp$ consists of a head \bar{x} and a body gp which is a graph pattern. The head of a query represents the answer variables of the query, which are a subset of all variables occurring in the body of q . We use $\text{Vars}(q)$ to refer to the set of all variables in a query and $\text{Head}(q)$ to refer to the answer variables. The body of a query consists of a graph pattern, which is either a conjunction of two patterns or a triple pattern where either subject, object or both have been

replaced by variables. They are connected via a path expression r . For simplicity, we restrict ourselves to path expressions consisting of either standard properties p representing an edge between one node to another or the inverse of a path expression r^- . As an example, consider a query that queries for all instances of `Student` that have at least `-hasFriend→` relation to an instance of `Person`. Figure 2.6 depicts this query as a conjunctive query using abstract syntax (left side) as well as the concrete SPARQL query.

$q_1 = x \leftarrow x \text{ type Student} \wedge$ $x \text{ hasFriend } y \wedge$ $y \text{ type Person}$	<pre> 1 SELECT ?x WHERE { 2 ?x rdf:type :Student. 3 ?x :hasFriend ?y. 4 ?y rdf:type :Person. 5 }</pre>
--	--

Figure 2.6.: SPARQL query in both abstract syntax (left) and standard SPARQL syntax (right).

Semantics Evaluation of queries requires the definition of a *mapping* μ . A mapping μ is a function $\mu : \text{Vars}(q) \rightarrow \text{Nodes}(G)$ mapping variables of the query q to nodes of the graph G . We use Ω to denote sets of mappings. The domain of μ is the set of variables that occur in q . Two mappings μ_1 and μ_2 are called *compatible* if for all $x \in \text{Dom}(\mu_1) \cap \text{Dom}(\mu_2)$, it is the case that $\mu_1(x) = \mu_2(x)$. That is, if a variable occurs in both mappings, then it must be mapped to the same value. Last, to model projection of a query answer μ onto the answer variables \bar{x} , we use $\mu|_{\bar{x}}$ to denote the restriction of μ to the domain \bar{x} . Evaluation of a query q over an RDF graph G , denoted as $\llbracket q \rrbracket_G$, can then be defined as follows (see Figure 2.5, right side): In case of a triple pattern, the answer is a mapping μ that maps variables onto graph nodes such that they occur in the graph (rules Q-SVAR, Q-OVAR and Q-VARS). We use the notation $r(G)$ to denote the evaluation of the path r —that is, the set of nodes (o, o') for which there is a connection via the path r in G . Conjunction of two graph patterns is evaluated by evaluating each pattern individually and then joining the results ($\llbracket gp_1 \rrbracket_G \bowtie \llbracket gp_2 \rrbracket_G$). Joining two sets of mappings $\Omega_1 \bowtie \Omega_2$ means taking the union of the individual mappings $\mu_1 \cup \mu_2$ for $\mu_1 \in \Omega_1$ and $\mu_2 \in \Omega_2$ provided that μ_1 and μ_2 are compatible. That is, if variables occurring in both evaluation results map to the same graph nodes. Projection then only requires the evaluation of the body and restriction of the resulting functions to the answer variables.

As an example, consider the evaluation of q_1 (see Figure 2.6) over the graph G_1 (see Figure 2.3), written $\llbracket q_1 \rrbracket_{G_1}$. First, the body of the query needs to be evaluated and the results will then be joined:

$$\llbracket x \text{ type Student} \rrbracket_{G_1} \bowtie \llbracket x \text{ hasFriend } y \rrbracket_{G_1} \bowtie \llbracket y \text{ type Person} \rrbracket_{G_1}$$

Evaluating each pattern yields sets of functions where variables `x` and `y` are mapped to actual graph

nodes:

$$\{\mu_1 = \{(x \mapsto \text{bob})\}\} \bowtie \{\mu_2 = \{(x \mapsto \text{bob}), (y \mapsto \text{alice})\}\} \bowtie \{\mu_3 = \{y \mapsto \text{bob}\}, \mu_4 = \{y \mapsto \text{alice}\}\}$$

Of those mappings, only μ_1 , μ_2 and μ_4 are compatible because they map the variable x to bob and the variable y to alice . This leaves μ_2 as the remaining mapping:

$$\{\mu_2 = \{(x \mapsto \text{bob}), (y \mapsto \text{alice})\}\}$$

Finally, function restriction $\mu_2|_x$ is used to restrict μ_2 to the answer variable x , leaving a single answer $\llbracket q_1 \rrbracket_{G_1} = \{(x \mapsto \text{bob})\}$ that maps the variable x to bob .

2.2.3. Description Logics

Ontologies provide rich conceptual schemas for the Semantic Web [53]. In this thesis, we focus on highly expressive ontology languages which are rooted in a family of knowledge representations formalisms known as *description logics (DL)*. Description logics separate the domain knowledge into two parts: the terminological part (T-Box) which constitutes a schema, as well as the instance data (assertional part or A-Box). The T-Box uses concept descriptions, also called concept expressions, to model the domain [86], e. g., the fact that a `Student` is a `Person`. The terminological knowledge can then be used to perform various reasoning tasks on the instance data, such as inferring that `bob` is a `Person`. A knowledge base is then a set of logical axioms containing both T-Box and A-Box assertions.

In the context of the Semantic Web, the W3C recommendation *Web Ontology Language (OWL)* [47] provides highly expressive ontology description languages. It is grounded in the description logic *SROIQ* [54]. For simplicity, we focus on a subset of the available syntactic constructs. The description logic used in the thesis is named *ALCOIQ* where the name is an acronym of its available constructors: *ALC* for the most commonly used *Attributive Language with Complements* extended with nominals (*O*), inverse role expressions (*I*) as well as qualified number restrictions (*Q*).³

Concept Expressions A description logic knowledge base K is comprised of a set of axioms. Axioms can either be terminological statements, belonging to the T-Box, or assertional statements belonging to the A-Box. Statements are constructed using a signature $Sig(K)$. The signature defines atomic elements of the knowledge base, which can then be combined into expressions using a range of available constructors. A signature of a knowledge base $Sig(K) = (N_A, N_P, N_O)$ is a triple consisting of a set of atomic concept names N_A (e. g., `Student`), a set of relation or property names N_P (e. g., `hasFriend`) and a set of atomic objects N_O (e. g., `bob`). Formal semantics of DL is based on

³A major difference between *ALCOIQ* and *SROIQ* is the latter allows for using an R-Box in which, for example, role inclusions or reflexivity of roles can be specified.

Constructor Name	Syntax	Semantics
atomic property	p	$p^I \subseteq \Delta^I \times \Delta^I$
inverse role	r^-	$\{(o', o) \mid (o, o') \in r^I\}$
atomic concept	A	$A^I \subseteq \Delta^I$
nominal concept	$\{o\}$	$\{o^I\}$
top	\top	Δ^I
negation	$\neg C$	$\Delta^I \setminus C^I$
conjunction	$C \sqcap D$	$C^I \cap D^I$
qualified number restriction	$\geq nr.C$	$\{o \mid \{o' \mid (o, o') \in r^I \wedge o' \in C^I\} \geq n\}$

Figure 2.7.: Syntax and Semantics of roles r and concept expressions C, D in the description logic \mathcal{ALCOIQ} .

first-order logic [18]. An interpretation I is a pair consisting of a non-empty universe Δ^I and an interpretation function \cdot^I . The universe “can be understood as the entirety of individuals or things which exist in the world that I represents” [84]. The interpretation function \cdot^I then maps each object $o \in N_O$ to an element of the universe $o^I \in \Delta^I$. Furthermore, the interpretation I assigns each atomic concept name $A \in N_A$ to a set $A^I \subseteq \Delta^I$ and each atomic property $p \in N_P$ to a binary relation $p^I \subseteq \Delta^I \times \Delta^I$.

Derived concept expressions

$\perp \stackrel{\text{def}}{=} \neg \top$ $C \sqcup D \stackrel{\text{def}}{=} \neg(\neg C \sqcap \neg D)$ $\{\bar{o}\} \stackrel{\text{def}}{=} \{o_1\} \sqcup \dots \sqcup \{o_n\}$	$\leq nr.C \stackrel{\text{def}}{=} \neg(\geq n + 1 r.C)$ $= nr.C \stackrel{\text{def}}{=} (\geq nr.C) \sqcap (\leq nr.C)$ $\forall r.C \stackrel{\text{def}}{=} \neg(\leq 0 r. \neg C)$ $\exists r.C \stackrel{\text{def}}{=} \geq 1 r.C$
--	--

Figure 2.8.: Concept expressions derived from concept expressions as defined in Figure 2.7.

More complex expressions can be built from these atomic elements as shown in Figure 2.7. Similar to path expressions as defined in SPARQL, role expressions, represented by the metavariable r , are either atomic properties p or the inverse of role expressions r^- . Concept expressions, represented by the metavariables C and D , are either atomic concepts (represented by the metavariable A), nominal concepts denoted by $\{o\}$ that are created by enumerating objects or \top to represent the set of all objects. Concept expressions can also be composed through negation $\neg C$, conjunction $C \sqcap D$ or

Name	Syntax	Semantics
concept inclusion	$C \sqsubseteq D$	$C^I \subseteq D^I$
concept equivalence	$C \equiv D$	$C^I = D^I$
concept assertion	$o : C$	$o^I \in C^I$
role assertion	$(o, o') : r$	$(o^I, o'^I) \in r^I$
object equivalence	$o \equiv o'$	$o^I = o'^I$

Figure 2.9.: Syntax and Semantics of axioms in the description logic \mathcal{ALCOIQ} .

qualified number restrictions $\geq nr.C$ expressing that there must be at least n successors via the role expression r that belongs to the concept C . A number of additional constructors can be derived from these basic ones (see Figure 2.8) such as existential quantification $\exists r.C$. For example, the set of everyone who studies at a university is expressed as $\exists \text{studiesAt.University}$. Other derived constructs include the bottom element \perp , disjunction $C \sqcup D$ universal quantification $\forall r.C$ as well as number restrictions requiring less than n successors $\leq nr.C$ and exactly n successors $=nr.C$.

Semantic statements Using the previously defined concept expressions, terminological and assertional statements (or axioms) can be defined (see Figure 2.9). A knowledge base K is a set of axioms consisting of terminological and assertional statements. Terminological axioms constitute the conceptualization of the data while assertional statements are the actual data. It is possible to express that two concept expressions are either equivalent $C \equiv D$ or in a subsumptive relationship $C \sqsubseteq D$. In terms of the actual data, objects can be an instance of a concept expression $o : C$, can be connected to another object via a role expression $(o, o') : r$ or be semantically equivalent even though they may be syntactically different $o \equiv o'$.

As an example, consider the knowledge base K_1 defined in Figure 2.10: Conceptually, instances of **Person** have a name (line 2). Likewise, instances of **University** have a location (line 3). Studying at a **University** means that one is a **Student** (line 4). A **Student** is a **Person** and all of his or her friends are also instances of **Student** (line 5). Lastly, a **Professor** works at a **University** (line 6). In terms of instance data, **uniko** is a **University** (line 8). **bob** studies at **uniko** (line 9). **bob** has a **hasFriend** relation pointing to **alice** (line 10). Lastly, **charlie** is an instance of **Professor** (line 11).

Several things are noteworthy in this example. First, the entirety of the information encoded in the graph G_1 shown in Figure 2.3 is contained in the example. Even though the A-Box statements **bob : Student** and **alice : Person**, which are equivalent to the RDF triples $(\text{bob})\text{-type}\rightarrow(\text{Student})$ and $(\text{alice})\text{-type}\rightarrow(\text{Person})$, are not explicitly mentioned, they follow logically from the example. Since **bob**

```

1 // Conceptualization / T-Box statements
2 Person  $\sqsubseteq$   $\exists$  hasName.  $\top$ 
3 University  $\sqsubseteq$   $\exists$  locatedIn.  $\top$ 
4  $\exists$  studiesAt. University  $\sqsubseteq$  Student
5 Student  $\sqsubseteq$  Person  $\sqcap$   $\forall$  hasFriend. Student
6 Professor  $\sqsubseteq$   $\exists$  worksAt. University
7 // Instance data / A-Box statements
8 uniko : University
9 (bob, uniko) : studiesAt
10 (bob, alice) : hasFriend
11 charlie : Professor

```

Figure 2.10.: Example knowledge base K_1 .

studies at `uniko` which is a `University`, he must be `Student` (c.f. line 4). Likewise, since `bob` is a `Student` and students can only be friends with other students, it follows that `alice` is also an instance of `Student` (c.f. line 5). As instances of `Student` are also instances of `Person`, it follows that `alice` is a `Person` (c.f. line 5). Second, even though `charlie` is a `Professor` and professors must work at a `University` (c.f. line 6), we do not know where `charlie` works. This does not constitute an error, but rather incomplete knowledge. Even though there is no syntactical element representing the `University`, we know that `charlie` works at one. Lastly, it must be pointed out that there is no unique name assumption in description logics. Unless otherwise specified, it may be that two syntactically different objects refer to the same object. As an example where this is useful, consider a changed name due to marriage. Then it may be the case that two syntactically different elements refer semantically to the same person. For example, by adding `bob \equiv charlie` to the knowledge base, it is enforced that `bob` and `charlie` are semantically the same object.

Logical consequence Logical consequence (entailment) is defined through interpretations. In a given interpretation I , a statement is either true or false. For a statement $stat$ built according to the syntax in Figure 2.9, we use the satisfaction relationship \models if its semantics constraint according to Figure 2.9 is true in an interpretation I (written $I \models stat$). An interpretation I satisfies a set of axioms $Stat$ if $\forall stat \in Stat : I \models stat$. An interpretation I that satisfies all axioms in both the T-Box and A-Box of a knowledge base K , written $I \models K$ is called a model of K . We use $\text{Mod}(K)$ to refer to the set of all models of K . A statement follows logically from a knowledge base K if it is true in all models of the knowledge base K .

Definition 3 (Entailment). Let K be a knowledge base, let $stat$ refer to a statement and let \mathcal{I} be the set of all possible interpretations. The statement $stat$ is entailed, written $K \models stat$, if $\forall I \in \mathcal{I} : I \models K \Rightarrow I \models stat$.

As an example, consider the knowledge base K_1 and the statement $bob : Student$ again. K_1 states that $bob^I \in (\exists studiesAt.University)^I$ for all models of K_1 due to bob studying at $uniko$ (lines 8–9). Line 4 defines that $(\exists studiesAt.University)^I \subseteq Student^I$ for all models of K_1 . It follows that $bob^I \in Student^I$ is true in all models. The statement $bob : Student$ is therefore a logical consequence of K_1 , written $K \models bob : Student$.

In general, one of the following three cases must be true for a statement:

1. A statement $stat$ can be true in all models of a knowledge base K . It is therefore a logical consequence of K , written $K \models stat$.
2. A statement $stat$ may be false in all models of a knowledge base K . Therefore, its negation is a logical consequence of K . For example, the statement $bob : \neg Student$ is false in all models of K_1 . Its negation, $bob : Student$ however is true in all models of K_1 .
3. A statement $stat$ may be true in some models, but false in other models. For example, the statement $bob : Professor$ is true in some models and false in others. We therefore do not know whether the statement is true or false.

Several standard reasoning tasks can be identified. For one, it is possible to ask for satisfiability—that is, whether a knowledge base K has at least one model. Closely related is the question of concept expression satisfiability. That is, asking whether a concept expression C is satisfiable with respect to a knowledge base K . This requires finding a model of K in which the interpretation of C is non-empty. Entailment of statements can be reduced to satisfiability tasks. For example, checking whether $Student \sqsubseteq Person$ is equivalent to showing that the concept expression $Student \sqcap \neg Person$ is unsatisfiable. Lastly, standard reasoning tasks also include instance retrieval as well as conjunctive query answering [84].

Relation to RDF and SPARQL In the Semantic Web technology stack, OWL ontologies are typically represented using RDF. The complete DL knowledge base K_1 as shown in Figure 2.10 can be mapped into an RDF data graph. In particular for A-Box statements, this translation is straightforward:

Name	DL Axiom	RDF data graph
concept assertion	$o : c$	
role assertion	$(o, o') : r$	

Objects (e.g., `bob`) in description logics are simply graph nodes in the RDF data graph (such as `(bob)`). Likewise, atomic concepts (e.g., `Student`) are represented as graph nodes (`(Student)`). Concept and role assertions then simply indicate relations between graph nodes. For example, `bob : Student` is represented as `(bob)-type→(Student)` whereas `(bob, alice) : hasFriend` is represented as `(bob)-hasFriend→(alice)`. Representation of more advanced concept expressions or other T-Box statements require the use of blank nodes. As an example, the axiom `Student ⊆ ∃ studiesAt.University` is represented as following (given in N3 notation):

```

1 :Student rdfs:subClassOf [
2   rdf:type owl:Restriction;
3   owl:onProperty :studiesAt;
4   owl:someValuesFrom :University;
5 ].

```

We avoid going into detail on the representation of terminological knowledge. The full translation can be found in [84]. Instead, we focus on the effects of terminological knowledge on an RDF data graph. Consider the small knowledge base K_2 as given in Figure 2.11 (left side). Its T-Box states that everyone who studies at a `University` is a `Student`. Furthermore, it states that a `Student` is also always a `Person`. The A-Box contains the facts that `bob` studies at `uniko` and that `uniko` is a `University`. We assume

```

1 // Conceptualization / T-Box
2 ∃ studiesAt.University ⊆ Student
3 Student ⊆ Person
4 // Instance data / A-Box
5 (bob, uniko) : studiesAt
6 uniko : University

```

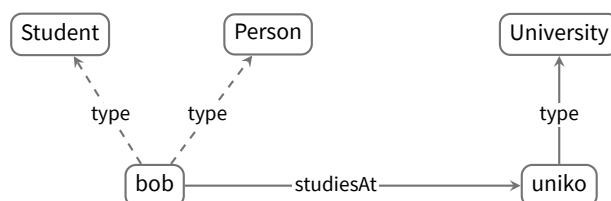


Figure 2.11.: A small knowledge base K_2 (left) and its RDF data graph (implicit knowledge in dashed lines).

that the RDF data graphs for knowledge bases (see Figure 2.11) are closed under logical consequence. That is, the RDF data graph does not only contain `(bob)-studiesAt→(uniko)`, but also `(bob)-type→(Student)` and `(bob)-type→(Person)`.

We omit details of querying a knowledge base with SPARQL as introduced in Section 2.2.2. Further information, including the official W3C entailment regimes for SPARQL can be found in [60, 45]. We assume that querying a knowledge base K with a query q means querying its representation as an RDF data graph that is closed under consequence. Real triplestores that support OWL ontologies, for example Stardog [11] or RDFox [7], work similarly. Evaluation of SPARQL queries will include implicit statements through a variety of implementation techniques—e.g., the materialization of implicit

relations such as $\text{bob} \text{--type--} \rightarrow \text{Person}$.

While we use $\llbracket q \rrbracket_G$ to indicate the evaluation of a query q over an RDF graph G , we write $\llbracket q \rrbracket_K$ for the evaluation of q over the knowledge base K . As an example, a query for all instances of Person over knowledge base K_2 yields bob as an answer:

$$\llbracket x \leftarrow x \text{ type Person} \rrbracket_{K_2} = \{\mu_1 = \{(x \mapsto \text{bob})\}\}$$

One key difference between querying a knowledge base and an RDF data graph must be mentioned: Models of a knowledge base may introduce anonymous objects that have no syntactic representation. Consider the knowledge base and RDF graph given in Figure 2.12. The knowledge base states that professors must have a `worksAt` relation pointing to an instance of `University` and that `charlie` is an instance of a `Professor`. To satisfy the knowledge base, interpretations must introduce an anonymous

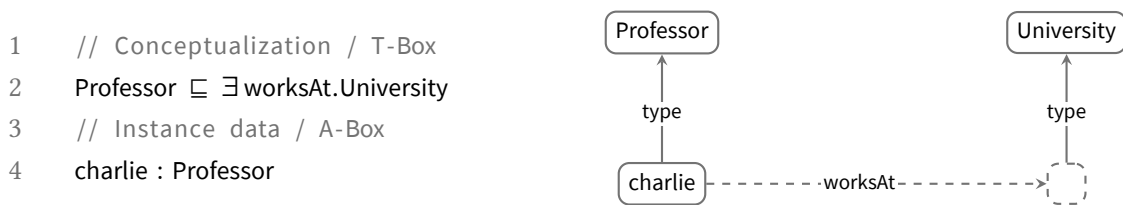


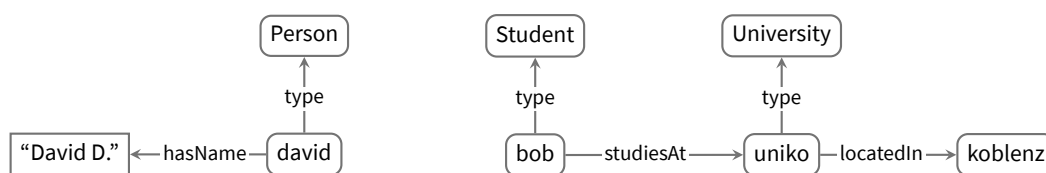
Figure 2.12.: A small knowledge base K_3 (left) and its RDF data graph (implicit knowledge in dashed lines).

object that represents the university where `charlie` works. While it is certain that it exists, it has no syntactic representation in the RDF data graph. Subsequently, in standard reasoner or triplestore implementations (e. g., Hermit [72] or Stardog [11]), such a node cannot be returned by the evaluation of a query.

2.2.4. Shape Constraint Language

The Shapes Constraint Language (SHACL) is a W3C standard for validating RDF graphs [59]—that is, testing whether the RDF graph conforms to a certain schema. SHACL differentiates between the *shape graph* that contains schematic definitions and the *data graph* that is being validated. A data graph is any RDF graph that is to be validated with respect to the shape graph. As an example, consider the data graph shown in Figure 2.13. `david` is an instance of `Person` and has a name. `bob`, an instance of `Student`, studies at `uniko`, which is an instance of `University`. The location of `uniko` is `koblenz`.

A shape graph consists of *shapes* that group *constraints* and provide so called *target nodes*. Target nodes specify which nodes of the data graph have to be valid with respect to the constraints. To exemplify this, consider a shape graph consisting of three shapes `StudentShape`, `PersonShape` and `UniversityShape` (see Figure 2.14). The shape `StudentShape` (lines 1–8) targets all instances of `Student` (line 2). Its constraints (lines 3–8) enforces that all instances of `Student` have at least one `–studiesAt→` relation (lines 4–5) and

Figure 2.13.: An example of an RDF data graph G_2 .

```

1  ex:StudentShape a sh:NodeShape;
2  sh:targetClass ex:Student;
3  sh:property [
4    sh:path ex:studiesAt;
5    sh:minCount 1;
6    sh:node ex:UniversityShape
7  ];
8  sh:class ex:Person.
9
10 ex:UniversityShape a sh:NodeShape;
11 sh:property [
12   sh:path ex:locatedIn;
13   sh:minCount 1 ].
14 ex:PersonShape a sh:NodeShape;
15 sh:targetClass ex:Person;
16 sh:property [
17   sh:path ex:hasName;
18   sh:minCount 1;
19   sh:maxCount 1
20 ].
21
22
23
24
25
26

```

Figure 2.14.: Example of a SHACL shape graph S_1 in N3 Notation.

that everything reachable via the --studiesAt-- relation is valid with respect to the UniversityShape (lines 4 and 6). Furthermore, students must also be instance of Person . The UniversityShape (lines 10–13) has no concrete targets, but requires nodes to have at least one --locatedIn-- relation (lines 12–13). Lastly, the PersonShape (lines 15–21) targets all instances of Person and enforces that each instance has exactly one --hasName-- relation (lines 18–20).

We use the metavariable S to denote a schema graph whereas \mathcal{S} denotes the set of all possible schema graphs. A SHACL validator essentially provides a function $\text{validate} : \mathcal{G} \times \mathcal{S} \rightarrow \{\text{true}, \text{false}\}$ that takes a data graph G and a schema graph S and determines whether the graph *conforms* to the schematic descriptions provided by the shape graph. In that, SHACL behaves similarly to validators for JSON Schema [5, 27] or XML Schema [42]. In case of the concrete example above, validation fails for G_2 and S_1 ($\text{validate}(G_2, S_1) = \text{false}$) due to the node bob (see Figure 2.15). Since bob is an instance of Student , he must also be an instance of Person . In addition, being an instance of Person then requires him to have a --hasName-- relation for G_2 to be valid with respect to S_1 . Alternatively, removing the fact that bob is an instance of Student would also make G_2 valid with respect to S_1 .

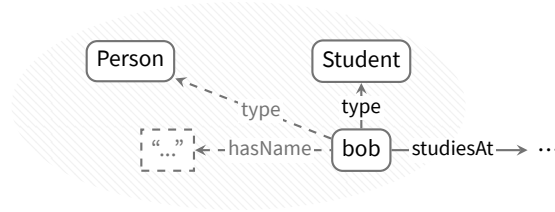


Figure 2.15.: Validation report highlighting erroneous area in the graph G_2 (missing nodes and edges in dashed lines).

As the official SHACL documentation [59] provides no formal semantics for the validation mechanism, in particular with respect to recursion, we leverage the formal semantics defined by Cormen et al. [35] which uses an abstraction that is based on first-order logic.

Constraints While shape graphs and subsequently constraints are typically given as RDF graphs, we use a logical abstraction. Constraints ϕ are constructed using the grammar as shown in Figure 2.16. \top represents a constraint that is always true, s references a shape name, o is a graph node, ρ is a path consisting of either a normal property p or an inverse of a path expression ρ^- and $n \in \mathbb{N}^+$. We use \mathcal{P} to indicate the set of all possible path expressions. As an example, consider a constraint expressing that a node must have at least one name. Such a constraint can be formulated as $\geq_1 \text{hasName}.\top$. A number of additional syntactic constructs can be defined in terms of the basic constructors. In particular, disjunction $\phi \vee \phi$, less-than $\leq_n \rho.\phi$ and equal-to $=_n \rho.\phi$ constraints as well as universal quantification $\forall \rho.\phi$ can be defined.

Evaluation of constraints may run into recursive cycles. As an example, assume a shape LocalShape whose constraint $\phi_{\text{LocalShape}}$ enforces that everything reachable via the $-\text{knows}->$ relation also conforms to LocalShape . Furthermore, assume an RDF data graph consisting of a single node who knows itself (see Figure 2.17). Validation of the node (b_1) must deal with this recursive cycle. The approach used for validation defines whether (b_1) conforms to LocalShape or not. In an optimistic approach, (b_1) is assumed to conform to the LocalShape unless proven otherwise. Evaluation of the constraint would succeed in this case as all nodes reachable from (b_1) via the $-\text{knows}->$ relation conform to the LocalShape . A pessimistic approach on the other hand does not assume that (b_1) conforms to LocalShape . Subsequently, it is not the case that everything reachable via the $-\text{knows}->$ relation conforms to LocalShape . Evaluation of the constraint fails and (b_1) would not conform to LocalShape .

We follow the proposal of [35] and introduce *assignments* to deal with this ambiguity. Similar to an interpretation in description logics, an assignment describes a possible world which is then used for evaluation. Subsequently, in case of the example in Figure 2.17 two possible worlds exist. One in which we assume (b_1) to conform to the LocalShape and one in which we do not. Formally, an assignment σ is a function assigning graph nodes o of an RDF data graph to a set of shape names s . Contrary to [35], we only consider total assignments that assign shape names to all nodes of the data graph.

Constraints (ϕ)

Syntax	Constraint Evaluation	$\llbracket \phi \rrbracket^{o,G,\sigma}$
$\phi ::=$	<i>constraint</i>	
\top	always true	$\llbracket \top \rrbracket^{o,G,\sigma} = \text{true}$
$ s$	shape reference	$\llbracket s \rrbracket^{o,G,\sigma} = \begin{cases} \text{true if } s \in \sigma(o) \\ \text{false otherwise} \end{cases}$
$ o$	graph node	$\llbracket o' \rrbracket^{o,G,\sigma} = \begin{cases} \text{true if } o = o' \\ \text{false otherwise} \end{cases}$
$ \phi \wedge \phi$	conjunction	$\llbracket \phi_1 \wedge \phi_2 \rrbracket^{o,G,\sigma} = \begin{cases} \text{true if } \llbracket \phi_1 \rrbracket^{o,G,\sigma} = \text{true and} \\ \quad \llbracket \phi_2 \rrbracket^{o,G,\sigma} = \text{true} \\ \text{false otherwise} \end{cases}$
$ \neg \phi$	negation	$\llbracket \neg \phi \rrbracket^{o,G,\sigma} = \begin{cases} \text{true if } \llbracket \phi \rrbracket^{o,G,\sigma} = \text{false} \\ \text{false otherwise} \end{cases}$
$ \geq_n \rho.\phi$	number restriction	$\llbracket \geq_n \rho.\phi \rrbracket^{o,G,\sigma} = \begin{cases} \text{true if } \{o' \mid (o, o') \in \rho(G) \text{ and} \\ \quad \llbracket \phi \rrbracket^{o',G,\sigma} = \text{true}\} \geq n \\ \text{false otherwise} \end{cases}$
$\rho ::=$	<i>path expression</i>	
$ p$	property	
$ \rho^-$	inverse path	
<i>New derived forms</i>		
$\phi_1 \vee \phi_2 \stackrel{\text{def}}{=} \neg(\neg\phi_1 \wedge \neg\phi_2)$		
$\leq_n \rho.\phi \stackrel{\text{def}}{=} \neg(\geq_{n+1} \rho.\phi)$		
$=_n \rho.\phi \stackrel{\text{def}}{=} (\leq_n \rho.\phi) \wedge (\geq_n \rho.\phi)$		
$\exists \rho.\phi \stackrel{\text{def}}{=} (\geq_1 \rho.\phi)$		
$\forall \rho.\phi \stackrel{\text{def}}{=} \neg(\leq_0 \rho.\neg\phi)$		

Figure 2.16.: Syntax and Evaluation rules of SHACL constraints.

Evaluation of whether a graph node conforms to a constraint then takes an assignment as a parameter and evaluates the constraint with respect to the given assignment.

Definition 4 (Total assignment). *Let G be an RDF graph with its set of nodes $\text{Nodes}(G)$ and S a set of shapes with its set of shape names $\text{Names}(S)$. An assignment σ is a total function $\sigma : \text{Nodes}(G) \rightarrow 2^{\text{Names}(S)}$. It maps graph nodes $o \in \text{Nodes}(G)$ to subsets of shape names. If a shape name $s \in \sigma(o)$, then o is assigned to the shape name s . For all $s \notin \sigma(o)$, the node o is not assigned to the shape s .*

Evaluating whether a graph node o in a given RDF graph G satisfies a constraint ϕ , written $\llbracket \phi \rrbracket^{o,G,\sigma}$, can then be defined as shown in Figure 2.16. Again, we use the notation $\rho(G)$ to denote the evaluation of the path ρ —that is, the set of nodes (o, o') for which there is a connectino

$$\phi_{\text{LocalShape}} = \forall \text{ knows. LocalShape}$$

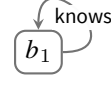


Figure 2.17.: An example of a problematic, recursive constraint definition and RDF graph G_{rec} .

via the path ρ in G . As an example, if G_{rec} denotes the graph in Figure 2.17 and σ_1 an assignment for which $\text{LocalShape} \in \sigma_1(b_1)$ is true, the constraint $\forall \text{ knows. LocalShape}$ evaluates to true $\llbracket \forall \text{ knows. LocalShape} \rrbracket^{b_1, G_{\text{rec}}, \sigma_1} = \text{true}$.

Shapes and Validation A shape is modelled as a triple (s, ϕ, \hat{q}) . It consists of a shape name s , a constraint ϕ and a query for target nodes \hat{q} . Target nodes indicate which nodes must fulfill the constraint in order for the graph to be valid. SHACL only allows for very specific queries when it comes to target nodes. Shapes can (1) have no targets at all, (2) explicitly enumerate target nodes, (3) have class-based targets, which targets all instances of a class, as well as either (4) subject-based or (5) object-based targets in which either all subjects or objects of a given relation are targeted (see Figure 2.18). We use \hat{q}_s to indicate that \hat{q} is the query for target nodes of shape s . Furthermore, in

$\hat{q} ::=$	<i>target node query</i>
\perp	no target
$ \ \{\bar{o}\}$	node-based target
$ \ x_1 \leftarrow x_1 \text{ type } \textit{class}$	class-based target
$ \ x_1 \leftarrow x_1 \text{ property } x_2$	subject-based target
$ \ x_1 \leftarrow x_2 \text{ property } x_1$	object-based target

Figure 2.18.: General form of target node queries in SHACL.

a slight abuse of notation, we write $o \in \llbracket \hat{q}_s \rrbracket_G$ to indicate that node o is a target node for shape s in graph G . In case of explicitly enumerated target nodes, we expect that all those node occur in G . We use S to represent a set of shapes. If $(s, \phi_s, \hat{q}_s) \in S$ and there is a s' appearing in ϕ_s , then we also assume that $(s', \phi_{s'}, \hat{q}_{s'}) \in S$. That is, we do not have references to shapes within a set that are undefined. To exemplify this, let us revisit the SHACL shape graph as given in Figure 2.14. Since StudentShape references UniversityShape in its constraints, the set of shapes must also contain the definition

of UniversityShape . The complete shape graph expressed as a set of shapes looks as follows:

$$S_1 = \{ (\text{StudentShape}, \geq_1 \text{ studiesAt. } \top \wedge \leq_0 \text{ studiesAt. } \neg \text{UniversityShape} \wedge \geq_1 \text{ type.Person,} \\ x \leftarrow x \text{ type Student}), \\ (\text{PersonShape}, =_1 \text{ hasName. } \top, x \leftarrow x \text{ type Person}), \\ (\text{UniversityShape}, \geq_1 \text{ hasLocation. } \top, \perp) \}$$

Intuitively, when validating an RDF graph with a set of shapes, only certain assignments are of interest. For one, if a node is a target node of a shape, then any sensible assignment should assign the shape to the node. That is, if graph G_2 as defined in Figure 2.13 is to be validated with the shapes defined in S_1 , then only assignments that assign StudentShape to bob should be considered. Second, if an assignment does assign a shape, then the constraint of that shape should evaluate to true. Such an assignment is called a *faithful assignment*.

Definition 5 (Faithful assignment). *An assignment σ for a graph G and a set of shapes S is faithful, iff for each $(s, \phi_s, \hat{q}_s) \in S$*

- $\forall o \in \text{Nodes}(G) : s \in \sigma(o) \Leftrightarrow \llbracket \phi_s \rrbracket^{o, G, \sigma}$.
- $\forall o \in \llbracket \hat{q}_s \rrbracket_G : s \in \sigma(o')$.

Validating an RDF graph means finding a faithful assignment. It is akin to satisfiability checking in Description Logics. It may not necessarily be possible to find a faithful assignment for a graph and a set of shapes. In particular, it may be that certain shapes prohibit the validation of any RDF graph—e. g., by using constraints such as $\neg \top$ in a shape. However, if a graph can be validated with a set of shapes, then the graph is said to *conform* to the set of shapes.

Definition 6 (Conformance). *An RDF graph G conforms to a set of shapes S iff there is at least one faithful assignment σ for G and S . We write $\text{Faith}(G, S)$ to denote the set of all faithful assignments for G and S .*

As an example, let us consider a version of G_2 in which the errors have been fixed. That is, bob is now not only a Student , but also a Person and has a name. A faithful assignment σ_1 then assigns StudentShape and PersonShape to bob whereas david is only assigned to PersonShape . Lastly, uniko is assigned to UniversityShape . Figure 2.19 shows the graph and the faithful assignment σ_1 .

Conformance and Target Nodes As it becomes important in later chapters of this thesis, we must point out the role of target nodes in the validation of an RDF graph. To exemplify target nodes, consider a set of SHACL shapes that only consists of PersonShape , but that does not have any target nodes:

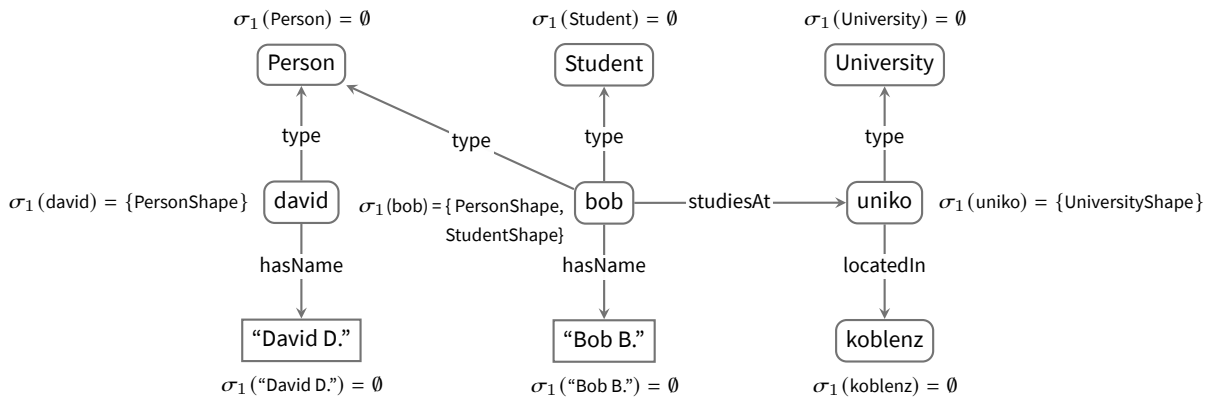


Figure 2.19.: RDF graph that is conformant to S_1 shown through the faithful assignment σ_1 .

$$S_2 = \{ (\text{PersonShape}, =_1 \text{hasName.T}, \perp) \}$$

Any RDF data graph is valid with respect to the set of shapes S_2 even though it may be that there are no nodes that conform to any shape. This is by design, as the SHACL definition specifically states

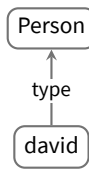


Figure 2.20.: RDF data graph G_3 that conforms to the set of shapes S_2 even though no nodes conform to any shapes.

that any graph is valid if no target nodes exist. For example, graph G_3 (see Figure 2.20) is valid with respect to the set of shapes S_2 because the assignment that assigns no shape to any node is faithful. Only once we modify S_2 such that PersonShape targets all instances of Person , graph G_3 becomes invalid.

Constraints in shapes in essence define a set of nodes. For example, the constraint $=_1 \text{hasName.T}$ of PersonShape in S_2 defines that PersonShape represents the set of nodes that have exactly one $-\text{hasName} \rightarrow$ relation. Queries for target nodes, such as \perp or $x \leftarrow x \text{ type Person}$, define the smallest set of nodes which must fulfill the constraint in order for the graph to be valid.

A Basic Programming Language (λ -calculus)

In order to define type-safe programming for RDF data, we rely on the λ -calculus as a basic programming language. This chapter defines syntax, semantics and typing rules of all necessary constructs. As common in programming language research, we avoid complexity by using a tiny core calculus that is extended with necessary features. We chose the λ -calculus over other calculi, e.g., object-calculi [12, 43], due to its simplicity.

We start with a simply typed λ -calculus with booleans, numerical values and let-bindings. We then extend it with recursion as well as records and lists as data structures. The resulting language, named λ_{Full} , is relatively small, yet it is Turing-complete and contains everything necessary for later chapters. It resembles functional programming languages such as OCaml or Haskell. The basic language design and definitions used in this chapter are taken from *Types and Programming Languages* [81] with slight modifications. The proof of soundness for λ_{Full} can be found in the appendix.

3.1. The Simply Typed λ -calculus with Subtyping

The λ -calculus is a formal system for expressing computations that solely relies on function definition and application. We start with the simply typed variant λ_{\rightarrow} , which will be extended to a version that supports subtyping named $\lambda_{<}$. The language introduced here is a simple call-by-value λ -calculus—that is, a λ -calculus that reduces terms to values before applying functions—which is already extended with booleans, numerical values and let-bindings in order to simplify examples.

3.1.1. Syntax and Semantics

Syntax Syntactically, the most fundamental terms (abbreviated as t) in λ_{\rightarrow} (see Figure 3.1) are variables x , λ -abstractions $\lambda x:T.t$ that represent functions, and function applications of the form $t t$. We use *explicitly typed function abstractions* that clearly state the type of their argument. That is, we only allow the syntactic form $\lambda x:T.t$ to define functions. In general, it is also possible to

allow for omitting the type, thus using the syntactic form $\lambda x . t$. The type of the argument x can be *reconstructed* based on the body of the function and how the function is used. We chose explicit type annotations as this allows us to focus on the actual process of type checking instead of the additional complexity introduced by reconstructing the type. Other constructs include the syntactic elements of the language \mathbb{NB} as defined in Section 2.1—that is, `true`, `false`, if-then-else statements as well as numerical values and the terms `iszero`, `succ` and `pred`. Lastly, we also include `let`-statements which simplifies the formulation of programs as they allow for defining symbols that represent other terms—e.g., a symbol that represents a function. Values (v) include primitive values (`true`, `false` and numerical values) and λ -abstractions. An important feature of the λ -calculus is that functions represented by λ -abstractions are values themselves. With regard to types, functions are typed using the function type constructor $T \rightarrow T$. Additionally, the language also provides the primitive types `Bool` and `Nat`.

As an example, consider a λ -abstraction representing a function that takes a term of type `Nat` and then checks whether that term is zero:

$$\lambda x : \text{Nat} . \text{iszero } x$$

This term has type `Nat \rightarrow Bool`. A term that applies this λ -abstraction to the term `0` is then a simple function application (represented through $t_1 t_2$)

$$(\lambda x : \text{Nat} . \text{iszero } x) 0$$

Lastly, a `let`-statement allows for introducing a symbol that represents the λ -abstraction. We can, for example, represent the λ -abstraction defined above through the symbol `f`:

$$\text{let } f = \lambda x : \text{Nat} . \text{iszero } x \text{ in } f 0$$

Defining functions with multiple arguments is possible by nesting λ -abstractions. This is known as *currying*. Intuitively, if a function has two arguments, applying the λ -abstraction to the first argument simply returns another λ -abstraction that is then applied to the second argument. As an example, a λ -abstraction `g` that takes two arguments, a value of type `Bool` and a value of type `Nat`, can be defined as follows:

$$\text{let } g = \lambda x : \text{Bool} . \lambda y : \text{Nat} . \text{if } x \text{ then } y \text{ else } 0 \text{ in } (g \text{ true}) 1$$

Semantics The operational semantics of λ_{\rightarrow} is defined in Figure 3.1. Most cases do not differ from the definitions used in \mathbb{NB} (see Figure 2.1). For example, if the guard (t_1) in `if t_1 then t_2 else t_3` is either `true` or `false`, then the term is reduced to either the `then`-branch (t_2) or `else`-branch (t_3) respectively (rules E-IFTRUE and E-IFFALSE).

λ_{\rightarrow}		$t \longrightarrow t'$
Syntax		Evaluation
$t ::=$	<i>terms:</i>	
x	variable	if true then t_2 else $t_3 \longrightarrow t_2$ (E-IFTRUE)
$ \text{true}$	constant true	if false then t_2 else $t_3 \longrightarrow t_3$ (E-IFFALSE)
$ \text{false}$	constant false	$\frac{t_1 \longrightarrow t'_1}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \longrightarrow \text{if } t'_1 \text{ then } t_2 \text{ else } t_3}$ (E-IF)
$ \text{if } t \text{ then } t \text{ else } t$	if-then-else	$\frac{t_1 \longrightarrow t'_1}{\text{succ } t_1 \longrightarrow \text{succ } t'_1}$ (E-SUCC)
$ 0$	constant zero	$\text{pred } 0 \longrightarrow 0$ (E-PREDZERO)
$ \text{succ } t$	successor	$\text{pred } (\text{succ } nv_1) \longrightarrow nv_1$ (E-PRESUCC)
$ \text{pred } t$	predecessor	$\frac{t_1 \longrightarrow t'_1}{\text{pred } t_1 \longrightarrow \text{pred } t'_1}$ (E-PRED)
$ \text{iszero } t$	zero test	$\text{iszero } 0 \longrightarrow \text{true}$ (E-ISZEROZERO)
$ \text{let } x = t \text{ in } t$	let binding	$\text{iszero } (\text{succ } nv_1) \longrightarrow \text{false}$ (E-ISZEROSUCC)
$ \lambda x : T . t$	abstraction	$\frac{t_1 \longrightarrow t'_1}{\text{iszero } t_1 \longrightarrow \text{iszero } t'_1}$ (E-ISZERO)
$ t t$	application	$\text{let } x = v_1 \text{ in } t_2 \longrightarrow [x \mapsto v_1]t_2$ (E-LETV)
$v ::=$	<i>values:</i>	$\frac{t_1 \longrightarrow t'_1}{\text{let } x = t_1 \text{ in } t_2 \longrightarrow \text{let } x = t'_1 \text{ in } t_2}$ (E-LET)
π	primitive value	$\frac{t_1 \longrightarrow t'_1}{t_1 t_2 \longrightarrow t'_1 t_2}$ (E-APP1)
$ \lambda x : T . t$	abstraction value	$\frac{t_2 \longrightarrow t'_2}{v_1 t_2 \longrightarrow v_1 t'_2}$ (E-APP2)
$\pi ::=$	<i>primitive values:</i>	
true	true value	
$ \text{false}$	false value	
$ nv$	numerical value	
$nv ::=$	numerical value	
0		
$ \text{succ } nv$		
$T ::=$	<i>types:</i>	
Π	types of primitives	
$ T \rightarrow T$	type of functions	
$\Pi ::=$	<i>primitive types:</i>	
Bool	type of booleans	
$ \text{Nat}$	type of natural numbers	
$\Gamma ::=$	<i>context:</i>	
\emptyset	empty context	
$\Gamma, x : T$	term variable binding	$(\lambda x : T . t_1) v_2 \longrightarrow [x \mapsto v_2]t_1$ (E-APPABS)

 Figure 3.1.: Simply typed λ -calculus (λ_{\rightarrow}) with booleans, numerical values and let bindings.

In case of `iszero t_1` , if the term t_1 is a numerical value, then either rule E-ISZEROZERO or rule E-ISZEROSUCC applies. Otherwise, if t_1 is not a value and can take a step, then it does (rule E-ISZERO). Function application `$t_1 t_2$` constitutes a new case. The evaluation relation first reduces the term t_1 to a value (rule E-APP1), then continues to reduce term t_2 (rule E-APP2). Lastly, the application itself can be performed (rule E-APPABS). Intuitively, application of a λ -abstraction `$\lambda x:T. t_1$` to a value v_2 means replacing all occurrences of the variable x in the function body t_1 with v_2 , denoted by $[x \mapsto v_2]t_1$. This is known as *substitution*. Substitution in general is intricate due to the naming of variables. As its details have no direct effect on this work, we only discuss the topic superficially and refer to [81] for a more in-depth discussion on the subject.

Definition 7 (Substitution). *Let x_1, x_2 be variables and t_1, t_2, t_3 be terms. A variable x_1 is said to be bound if it occurs in the body t_1 of a λ -abstraction `$\lambda x_1:T. t_1$` . A variable is free if it is not bound. Substitution of x_1 by t_2 in t_1 , denoted by $[x_1 \mapsto t_2]t_1$, is then defined as follows:*

$$\begin{aligned}
 [x_1 \mapsto t_2]x_1 &= t_2 \\
 [x_1 \mapsto t_2]x_2 &= x_2 && \text{if } x_2 \neq x_1 \\
 [x_1 \mapsto t_2](\lambda x_2:T. t_1) &= \lambda x_2:T. [x_1 \mapsto t_2]t_1 && \text{if } x_2 \neq x_1 \text{ and} \\
 &&& x_2 \text{ is not a free variable in } t_2 \\
 [x_1 \mapsto t_2]t_1 t_3 &= ([x_1 \mapsto t_2]t_1) ([x_1 \mapsto t_2]t_3) \\
 [x_1 \mapsto t_2]true &= true \\
 [x_1 \mapsto t_2>false &= false \\
 [x_1 \mapsto t_2]if t_1 &= if [x_1 \mapsto t_2] t_1 \\
 &\quad \text{then } t_3 && \text{then } [x_1 \mapsto t_2] t_3 \\
 &\quad \text{else } t_4 && \text{else } [x_1 \mapsto t_2]t_4 \\
 [x_1 \mapsto t_2]0 &= 0 \\
 [x_1 \mapsto t_2]succ t_1 &= succ [x_1 \mapsto t_2]t_1 \\
 [x_1 \mapsto t_2]pred t_1 &= pred [x_1 \mapsto t_2]t_1 \\
 [x_1 \mapsto t_2]iszero t_1 &= iszero [x_1 \mapsto t_2]t_1 \\
 [x_1 \mapsto t_2]let x_2=t_1 in t_3 &= let x_2=[x_1 \mapsto t_2]t_1 in [x_1 \mapsto t_2]t_3 && \text{if } x_2 \neq x_1 \text{ and} \\
 &&& x_2 \text{ is not a free variable in } t_2
 \end{aligned}$$

Consider the term `$\lambda x:\text{Bool}. y x$` where `$\lambda x:\text{Bool}$` constitutes the head of the λ -abstraction and `$x y$` its body. Applying the λ -abstraction to a value v_2 (c. f. rule E-APPABS) means substituting the variable x for v_2 in the body of the λ -abstraction, written $[x \mapsto v_2] x y$. However, the definition used in Definition 7 only works with terms “up to renaming of bound variables” (also known as *alpha conversion*) [81]. That is, we do not deal with cases in which variables would need to be renamed before substitution in order to preserve the meaning of a term. Reconsider the term `$\lambda x:\text{Bool}. y x$` . The variable x is of type `Bool` whereas the free variable y must represent a function. In case y is substituted for another symbol x , written $[y \mapsto x] (\lambda x:\text{Bool}. y x)$, the meaning of the term changes

as the body of the λ -abstraction now attempts to apply a value of type `Bool` to the same value. An approach that considered alpha conversion would need to rename `x` in the λ -abstraction `$\lambda x:\text{Bool} . y x$` to a different name, for example `z` to preserve the original meaning of the term. Thus, the substitution becomes `$[y \mapsto x] (\lambda z:\text{Bool} . y z)$` . We acknowledge this by adding a side condition to the definition of substitution on λ -abstractions (see Definition 7), but do not resolve the issue for simplicities sake. We simply assume that the names of all bound variables are unique. In practice, it is common to rely on a nameless representation of λ -abstractions known as *de Bruijn indices* to avoid the issue.

To highlight substitution, let us consider the term `$(\lambda x:\text{Nat} . \text{iszero } x) 0$` again. For this term, evaluation rule `E-APPABS` applies, meaning that the symbol `x` is substituted for the term `0` in the body of the λ -abstraction:

$$(\lambda x:\text{Nat} . \text{iszero } x) 0 \longrightarrow [x \mapsto 0]\text{iszero } x = \text{iszero } 0$$

Substitution of `x` for `0` in `iszero x` then yields the term `iszero 0` which evaluates to true using rule `E-ISZEROZERO`. In case of the `let`-statement which uses the symbol `f` to represent the λ -abstraction, rule `E-LETV` applies which substitutes `f` for the actual λ -abstraction:

$$\text{let } f = \lambda x:\text{Nat} . \text{iszero } x \text{ in } f 0 \longrightarrow [f \mapsto (\lambda x:\text{Nat} . \text{iszero } x)]f 0 = (\lambda x:\text{Nat} . \text{iszero } x) 0$$

Likewise, evaluation of the λ -abstraction represented by the symbol `g` which served as an example for a function with two parameters first substitutes the symbol `g` for the actual λ -abstractions via rule `E-LETV`:

$$\begin{array}{l} \text{let } g = \lambda x:\text{Bool} . \\ \quad \lambda y:\text{Nat} . \\ \quad \quad \text{if } x \text{ then } y \text{ else } 0 \\ \text{in } ((g \text{ true}) 1) \end{array} \longrightarrow \begin{array}{l} ((\lambda x:\text{Bool} . \\ \quad \lambda y:\text{Nat} . \\ \quad \quad \text{if } x \text{ then } y \text{ else } 0) \text{ true}) 1 \end{array}$$

The λ -abstraction can be applied to the term `true` (via rules `E-APP1` and `E-APPABS`), yielding again a function application:

$$((\lambda x:\text{Bool} . \lambda y:\text{Nat} . \text{if } x \text{ then } y \text{ else } 0) \text{ true}) 1 \longrightarrow (\lambda y:\text{Nat} . \text{if } \text{true} \text{ then } y \text{ else } 0) 1$$

Again, rule `E-APPABS` applies, substituting the symbol `y` for the term `1`:

$$(\lambda y:\text{Nat} . \text{if } \text{true} \text{ then } y \text{ else } 0) 1 \longrightarrow \text{if } \text{true} \text{ then } 1 \text{ else } 0$$

This term then evaluates via rule `E-IFTRUE` to the value `1`:

$$\text{if } \text{true} \text{ then } 1 \text{ else } 0 \longrightarrow 1$$

λ_{\rightarrow} (typed)	Extends λ_{\rightarrow} (Fig. 3.1)
<div style="display: flex; justify-content: space-between; align-items: center;"> <div style="margin-bottom: 10px;"><i>Typing rules</i></div> <div style="border: 1px solid black; padding: 2px;">$\Gamma \vdash t : T$</div> </div> <div style="margin-bottom: 10px;"> $\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad (\text{T-VAR})$ </div> <div style="margin-bottom: 10px;"> $\Gamma \vdash \text{true} \quad (\text{T-TRUE})$ </div> <div style="margin-bottom: 10px;"> $\Gamma \vdash \text{false} \quad (\text{T-FALSE})$ </div> <div style="margin-bottom: 10px;"> $\frac{\Gamma \vdash t_1 : \text{Bool} \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T} \quad (\text{T-IF})$ </div> <div style="margin-bottom: 10px;"> $\Gamma \vdash 0 : \text{Nat} \quad (\text{T-ZERO})$ </div> <div style="margin-bottom: 10px;"> $\frac{\Gamma \vdash t_1 : \text{Nat}}{\Gamma \vdash \text{succ } t_1 : \text{Nat}} \quad (\text{T-SUCC})$ </div>	<div style="margin-bottom: 10px;"> $\frac{\Gamma \vdash t_1 : \text{Nat}}{\Gamma \vdash \text{pred } t_1 : \text{Nat}} \quad (\text{T-PRED})$ </div> <div style="margin-bottom: 10px;"> $\frac{\Gamma \vdash t_1 : \text{Nat}}{\Gamma \vdash \text{iszero } t_1 : \text{Bool}} \quad (\text{T-ISZERO})$ </div> <div style="margin-bottom: 10px;"> $\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \text{let } x = t_1 \text{ in } t_2 : T_2} \quad (\text{T-LET})$ </div> <div style="margin-bottom: 10px;"> $\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 . t_2 : T_1 \rightarrow T_2} \quad (\text{T-ABS})$ </div> <div style="margin-bottom: 10px;"> $\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 t_2 : T_{12}} \quad (\text{T-APP})$ </div>

Figure 3.2.: Typing rules for the simply typed λ -calculus (λ_{\rightarrow}).

3.1.2. Type System

As with the language \mathbb{NB} , a type system can be defined for λ_{\rightarrow} that allows for avoiding run-time errors. The type system (see Figure 3.1) contains the primitive types `Bool` and `Nat` that are already known from \mathbb{NB} . The newly added functions are represented by the type constructor $T \rightarrow T$. Again, a function mapping a `Nat` to a term of type `Bool`, e. g., `$\lambda x : \text{Nat} . \text{iszero } x$` , is represented through the type `$\text{Nat} \rightarrow \text{Bool}$` . As mentioned before, we use explicitly typed λ -abstractions. In an explicitly typed λ -abstraction `$\lambda x : T_1 . t_1$` , the type of the argument x is T_1 . The return type of the λ -abstraction is reconstructed from the body t_1 using the fact that the argument x has type T_1 . This requires a way to store these type assignments. For this, a *typing context* or *typing environment* Γ is used. Subsequently, the typing relation $t : T$ becomes $\Gamma \vdash t : T$ where Γ is a set of assumptions about the types of free variables in t [81]. A typing context Γ is a sequence of variables and their types. We use \emptyset to denote the empty context whereas we use $\Gamma, x : T$ to denote the context Γ extended with a new binding $x : T$. We use $x : T \in \Gamma$ to denote that Γ stores the assumption that x has type T in the context Γ .

The typing rules (see Figure 3.2) can then be defined as follows: Variables are typed by using the information stored in the context (rule T-VAR). Rules for booleans, if-then-else constructs and natural numbers are the same as in \mathbb{NB} (see Figure 2.1). `true` and `false` are assigned to the type `Bool` (rules T-TRUE and T-FALSE). In case of an if-then-else expression, the guard must be of type `Bool` whereas both branches of the expression must have the same type (rule T-IF). The term `0` can again be directly typed (rule T-ZERO). Successors (`succ t_1`), predecessors (`pred t_1`) and testing for zeroes (`iszero t_1`) are

typed by checking whether t_1 is a `Nat` and subsequently assigning either `Nat` or `Bool` (rules `T-SUCC`, `T-PRED` and `T-ISZERO`). In case of a λ -abstraction $\lambda x : T_1 . t_2$, the binding $x : T_1$ is added to the context when typing the body of the function t_2 . When the body of the function has type T_2 , then the type of the λ -abstraction is $T_1 \rightarrow T_2$ (rule `T-ABS`). Function application $(t_1 t_2)$ requires t_1 to be a function type $T_{11} \rightarrow T_{12}$ whereas the type of t_2 must be the same (T_{11}) as the domain type of the function t_1 (rule `T-APP`). Lastly, in let-statements $(\text{let } x = t_1 \text{ in } t_2)$, first t_1 is assigned a type T_1 which is then assigned to the typing context Γ when assigning a type to t_2 (rule `T-LET`).

As an example, consider the λ -abstraction $\lambda x : \text{Nat} . \text{iszero } x$ again. This term can be typed with `Nat \rightarrow Bool`. First, rule `T-ABS` is applied which adds the variable binding $(x : \text{Nat})$ to the context. The body of the λ -abstraction is typed using rules `T-ISZERO` and `T-VAR`. This is shown using the following derivation tree:

$$\frac{\frac{\frac{(x : \text{Nat}) \in \emptyset, (x : \text{Nat})}{\emptyset, (x : \text{Nat}) \vdash x : \text{Nat}} \text{T-VAR}}{\emptyset, (x : \text{Nat}) \vdash \text{iszero } x : \text{Bool}} \text{T-ISZERO}}{\emptyset \vdash \lambda x : \text{Nat} . \text{iszero } x : \text{Nat} \rightarrow \text{Bool}} \text{T-ABS}$$

Applying the λ -abstraction to a term belonging to its domain type `Nat` will result in a term belonging to its co-domain type `Bool`. Consider the following derivation tree that shows the λ -abstraction being applied to the term `0`:

$$\frac{\frac{\vdots}{\emptyset \vdash \lambda x : \text{Nat} . \text{iszero } x : \text{Nat} \rightarrow \text{Bool}} \quad \frac{}{\emptyset \vdash 0 : \text{Nat}} \text{T-ZERO}}{\emptyset \vdash (\lambda x : \text{Nat} . \text{iszero } x) 0 : \text{Bool}} \text{T-APP}$$

Lastly, typing the let-statement that introduces a variable `f` for the λ -abstraction looks as follows:

$$\frac{\frac{\vdots}{\emptyset \vdash \lambda x : \text{Nat} . \text{iszero } x : \text{Nat} \rightarrow \text{Bool}} \quad \frac{\frac{(f : \text{Nat} \rightarrow \text{Bool}) \in \emptyset, f : \text{Nat} \rightarrow \text{Bool}}{\emptyset, f : \text{Nat} \rightarrow \text{Bool} \vdash f : \text{Nat} \rightarrow \text{Bool}} \text{T-VAR} \quad \frac{}{\dots \vdash 0 : \text{Nat}} \text{T-ZERO}}{\emptyset, f : \text{Nat} \rightarrow \text{Bool} \vdash f 0 : \text{Bool}} \text{T-APP}}{\emptyset \vdash \text{let } f = \lambda x : \text{Nat} . \text{iszero } x \text{ in } f 0 : \text{Bool}} \text{T-LET}$$

Algorithmic Type Checking Algorithmic type checking for the type system of λ_{\rightarrow} is straightforward as all rules are purely syntax driven. A typical implementation introduces a recursive function `TYPEOF` that takes a term t , a context Γ that is initially empty, and returns a type T (see Algorithm 1). The function introduces one case per rule as described in Figure 3.2. If no rule matches, a type error is raised.

Algorithm 1 Definition of `TYPEOF` for λ_{\rightarrow} .

```

function TYPEOF( $t, \Gamma$ )
  match  $t$  with
    case  $x$  when  $(x : T) \in \Gamma$  then  $T$ 
    case true then Bool
    case false then Bool
    case if  $t_1$  then  $t_2$  else  $t_3$  when TYPEOF( $t_1, \Gamma$ ) is Bool and TYPEOF( $t_2, \Gamma$ ) is  $T$ 
      and TYPEOF( $t_3, \Gamma$ ) is  $T$  then  $T$ 
    case 0 then Nat
    case succ  $t_1$  when TYPEOF( $t_1, \Gamma$ ) is Nat then Nat
    case pred  $t_1$  when TYPEOF( $t_1, \Gamma$ ) is Nat then Nat
    case iszero  $t_1$  when TYPEOF( $t_1, \Gamma$ ) is Nat then Bool
    case let  $x = t_1$  in  $t_2$  when TYPEOF( $t_1, \Gamma$ ) is  $T_1$  and TYPEOF( $t_2, (\Gamma, (x : T_1))$ ) is  $T_2$  then  $T_2$ 
    case  $\lambda x : T_1 . t_2$  when TYPEOF( $t_2, (\Gamma, (x : T_1))$ ) is  $T_2$  then  $T_1 \rightarrow T_2$ 
    case  $t_1 t_2$  when TYPEOF( $t_1, \Gamma$ ) is  $T_{11} \rightarrow T_{12}$  and TYPEOF( $t_2, \Gamma$ ) is  $T_{11}$  then  $T_{12}$ 
    case _ then FAIL
  
```

3.1.3. Subtyping

The type system as defined so far is relatively strict. In particular, rule T-APP enforces that the domain type of a λ -abstraction and the type of the value to which it is applied are the same:

$$\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 t_2 : T_{12}} \quad (\text{T-APP})$$

Subtyping eases that constraint in the sense that a type must not necessarily be equal, but can also be more special than the domain type. If a type is a set of values, then the idea is that a type T being a subtype of a type T' means that the set T is a subset of the set T' . Likewise, if T_{11} is the domain of a function, then any T which is a subtype of T_{11} should also be applicable since it means that T is a subset of T_{11} and all values of T must therefore also be values of T_{11} .

The addition of a subtyping relation, usually written $<:$, to λ_{\rightarrow} allows us to formalize this notion, leading to language called $\lambda_{<}$: (see Figure 3.3). Syntactically, we add a type that encompasses all other types called `Top`. We also add a new rule to the type system that formalizes the notion as described before—if a term t can be assigned the type T and T is a subtype of T' , written $T <: T'$, then term t must also have type T' (rule T-SUB). The remainder are the rules for subtyping. Every type is a subtype of itself, written $T <: T$ (rule S-REFL). If T is a subtype of type T' and T' is a subtype of type T'' , then T must also be a subtype of type T'' (rule S-TRANS). Every type is a subtype of `Top` (rule S-TOP). Lastly,

$\lambda_{<}$:		Extends λ_{\rightarrow} (see. Fig. 3.1 and 3.2)
<p><i>New syntactic forms</i></p> $T ::= \dots$ <p style="text-align: center;"> Top</p>	<p style="text-align: right;"><i>types:</i></p> <p style="text-align: center;">maximum type</p>	<p><i>Subtyping rules</i></p> <div style="border: 1px solid black; padding: 2px; display: inline-block; margin-bottom: 10px;">$T <: T'$</div> $T <: T \quad (\text{S-REFL})$ $\frac{T <: T' \quad T' <: T''}{T <: T''} \quad (\text{S-TRANS})$ $T <: \text{Top} \quad (\text{S-TOP})$ $\frac{T'_1 <: T_1 \quad T_2 <: T'_2}{T_1 \rightarrow T_2 <: T'_1 \rightarrow T'_2} \quad (\text{S-ARROW})$
<p><i>New typing rules</i></p> $\frac{\Gamma \vdash t : T \quad T <: T'}{\Gamma \vdash t : T'} \quad (\text{T-SUB})$	<div style="border: 1px solid black; padding: 2px; display: inline-block; margin-bottom: 10px;">$\Gamma \vdash t : T$</div>	

Figure 3.3.: Simply typed λ -calculus with subtyping ($\lambda_{<}$).

function types are in a subtype relation if their domains are in a flipped subtyping relationship (known as contra-variance) and their co-domains are in a subtyping relationship (known as co-variance).

As an example for subtyping, consider a λ -abstraction $\lambda x : \text{Top} . x$ with a parameter of type `Top` and which simply returns the given term. This λ -abstraction can be applied to a term of type `Bool` as rules T-SUB and S-TOP ensure that all terms of type `Bool` are also of type `Top`:

$$\frac{\begin{array}{c} \vdots \\ \emptyset \vdash \lambda x : \text{Top} . x : \text{Top} \rightarrow \text{Top} \end{array} \quad \frac{\frac{\emptyset \vdash \text{true} : \text{Bool}}{\emptyset \vdash \text{true} : \text{Top}} \text{T-TRUE} \quad \frac{}{\text{Bool} <: \text{Top}} \text{S-TOP}}{\emptyset \vdash \text{true} : \text{Top}} \text{T-SUB}}{\emptyset \vdash (\lambda x : \text{Top} . x) \text{true}} \text{T-APP}$$

Likewise, the λ -abstraction can be applied to a term of type `Nat`:

$$\frac{\begin{array}{c} \vdots \\ \emptyset \vdash \lambda x : \text{Top} . x : \text{Top} \rightarrow \text{Top} \end{array} \quad \frac{\frac{\emptyset \vdash 0 : \text{Nat}}{\emptyset \vdash 0 : \text{Top}} \text{T-ZERO} \quad \frac{}{\text{Bool} <: \text{Top}} \text{S-TOP}}{\emptyset \vdash 0 : \text{Top}} \text{T-SUB}}{\emptyset \vdash (\lambda x : \text{Top} . x) 0} \text{T-APP}$$

However, the type `Top` represents the most general set possible. λ -abstractions using more specific types as their domain type cannot be applied to the terms of type `Top` anymore. For example, for the term $(\lambda x : \text{Nat} . x) ((\lambda y : \text{Top} . y) 0)$, no derivation tree can be found. The term first applies a λ -abstraction of type `Top \rightarrow Top` to a term of type `Nat`. It tries to apply a λ -abstraction of type `Nat \rightarrow Nat` to the result of the first application. Intuitively, this code contains no errors as both λ -

abstractions simply return the term given to them. However, the type system will reject the term as being ill-typed.

Programming languages typically support down-casting of terms to assign more specific types. This is an error-prone operation as errors only surface at run-time. Our language does not support down-casting since it has no effect on the type systems defined in this thesis. Nevertheless, this highlights an important issue: type systems should assign the *principle type* to terms. That is, the most specific type available for a given term. A type system that does not assign the principle type but rather a more general type will reject more terms as ill-typed. As such, it is stricter without providing any benefits.

Algorithmic Type Checking The addition of rule τ -SUB introduces several changes to the function `TYPEOF` (see Algorithm 2). The cases for function application as well as if-then-else expressions must be revisited. Other cases remain as before. Function application $t_1 t_2$ so far required that the type of t_2 is exactly the domain type of t_1 . Instead, we now introduce a predicate `SUBTYPE` that determines whether t_2 is a subtype of the domain type of t_1 . Furthermore, we introduce a function `LUB` (discussed in Algorithm 3) that allows for constructing the least upper bound of two types. The

Algorithm 2 Definition of `TYPEOF` and `SUBTYPE` for $\lambda_{<}$.

```

function TYPEOF( $t, \Gamma$ )
  match  $t$  with
    case if  $t_1$  then  $t_2$  else  $t_3$  when TYPEOF( $t_1, \Gamma$ ) is Bool and TYPEOF( $t_2, \Gamma$ ) is  $T_2$ 
      and TYPEOF( $t_3, \Gamma$ ) is  $T_3$  then LUB( $T_2, T_3$ )
    case  $t_1 t_2$  when TYPEOF( $t_1, \Gamma$ ) is  $T_{11} \rightarrow T_{12}$  and TYPEOF( $t_2, \Gamma$ ) is  $T_2$ 
      and SUBTYPE( $T_2, T_{11}$ ) is true then  $T_{12}$ 
    case ... then ...       $\triangleright$  existing cases
    case _ then FAIL

function SUBTYPE( $T, T'$ )
  match ( $T, T'$ ) with
    case ( $T, T$ ) then true
    case ( $T, \text{Top}$ ) then true
    case ( $T_{11} \rightarrow T_{12}, T'_{11} \rightarrow T'_{12}$ ) when SUBTYPE( $T'_{11}, T_{11}$ ) and SUBTYPE( $T_{12}, T'_{12}$ ) then true
    case _ then false

```

`SUBTYPE` predicate is a straightforward translation of the subtyping rules in $\lambda_{<}$. (see Figure 3.3). The function introduces one case each for rules `S-REFL`, `S-TOP` and `S-ARROW`. At this point, rule `S-TRANS` has no effect yet. Furthermore, if-then-else expressions originally required both, the then-branch and the

else-branch, to be of the same type. Instead, we now introduce a function `LUB` that represents the least upper bound of two types (see Algorithm 3). In case of if-then-else expressions, the function `TYPEOF` can then simply return the least upper bound of both branches. Construction of the least upper bound

Algorithm 3 Definition of `LUB` and `GLB` for $\lambda_{<}$.

```

function LUB( $T, T'$ )
  match ( $T, T'$ ) with
    case ( $T, T'$ ) when SUBTYPE( $T, T'$ ) then  $T'$ 
    case ( $T_{11} \rightarrow T_{12}, T'_{11} \rightarrow T'_{12}$ ) then GLB( $T_{11}, T'_{11}$ )  $\rightarrow$  LUB( $T_{12}, T'_{12}$ )
    case _ then Top

function GLB( $T, T'$ )
  match ( $T, T'$ ) with
    case ( $T, T$ ) then  $T$ 
    case ( $T_{11} \rightarrow T_{12}, T'_{11} \rightarrow T'_{12}$ ) then LUB( $T_{11}, T'_{11}$ )  $\rightarrow$  GLB( $T_{12}, T'_{12}$ )
    case _ then FAIL

```

`LUB` is defined as follows: If one type is a subtype of the other, then the supertype acts as a least upper bound. For functions, we require an additional definition for the greatest lower bound `GLB`. The domain type of the newly constructed function is the greatest lower bound of the domain types of the input functions. The co-domain of the newly constructed function is the least upper bound of the co-domain types of the input functions. If no case of `LUB` matches, then the least upper bound must be the type `Top`. For the definition of the greatest lower bound `GLB`, it holds again that there is nothing to compute if the input are two syntactically equal types. For functions, the relation of domain and co-domain flips—the domain of the newly created function is the least upper bound of the domains of the input function. The co-domain of the newly created function is the greatest lower bound of the two input functions. Lastly, the computing the greatest lower bound can fail—if no case matches, then it is impossible to compute a greatest lower bound.

3.2. Extensions to the language

We now extend the language $\lambda_{<}$ with several features that are typically present in real programming languages. In particular, we extend the language with recursion, records for grouping data, as well as lists.

$\lambda_{\text{Recursion}}$		Extends $\lambda_{<}$ -calculus (Fig. 3.3)	
<i>New syntactic elements</i>		<i>New typing rules</i> $\boxed{\Gamma \vdash t : T}$	
$t ::= \dots$	<i>terms:</i>	$\frac{\Gamma \vdash t_1 : T_1 \rightarrow T_1}{\Gamma \vdash \mathbf{fix} \ t_1 : T_1} \quad (\text{T-FIX})$	
$\mathbf{fix} \ t$	fixed point of t		
<i>New evaluation rules</i> $\boxed{t \rightarrow t'}$		<i>New derived forms</i>	
$\mathbf{fix} \ (\lambda x : T_1 . t_2) \rightarrow [x \mapsto (\mathbf{fix}(\lambda x : T_1 . t_2))]t_2 \quad (\text{E-FIX1})$		$\mathbf{letrec} \ x : T_1 = t_1 \ \mathbf{in} \ t_2$	
$\frac{t_1 \rightarrow t'_1}{\mathbf{fix} \ t_1 \rightarrow \mathbf{fix} \ t'_1} \quad (\text{E-FIX2})$		$\stackrel{\text{def}}{=} \mathbf{let} \ x = \mathbf{fix}(\lambda x : T_1 . t_1) \ \mathbf{in} \ t_2$	

Figure 3.4.: Fixpoint operator for recursion.

3.2.1. Recursion

An important feature in general programming languages is the definition of recursive functions. That is, functions that call themselves. Importantly, recursion cannot be implemented using a simple `let`-statement such as `let f = t1 in t2`. Intuitively, typing of such a statement must fail as `f` is used in `t1` although the type of `f` is not yet known (rule T-LET). Instead, recursion requires the introduction of a fixed point combinator (see Figure 3.4). The term `fix t` constitutes the fixed point combinator that takes another term `t`. This must be a function mapping a value of `T1` onto a value of the same type (rule E-FIX1). If it is possible to reduce the term `t`, then the evaluation rules will do so (rule E-FIX2). Otherwise, if the term `t` is a function, the variable `x` is substituted for the fixed point of `t` in the body of the function. There are no additions to the subtyping rules. The procedure `TYPEOF` uses an additional case which is a straightforward implementation of T-FIX. As common in typed functional languages, a `letrec`-statement is introduced to simplify the usage of the fixed point combinator. As an example, consider a function `iseven` that takes a value of type `Nat` and returns either `true` if the value is even or `false` otherwise:

```

1  letrec iseven : Nat → Bool = λx : Nat .
2    if iszero x then true
3    else if iszero (pred x) then false
4    else iseven (pred (pred x))
5  in iseven 2

```


Evaluation of the program first applies rule E-LETV:

<pre> letrec iseven : Nat → Bool = λx : Nat. if iszero x then true else if iszero (pred x) then false else iseven (pred (pred x)) in iseven 2 </pre>	→	<pre> fix (λ iseven : Nat → Bool. λ x : Nat. if iszero x then true else if iszero (pred x) then false else iseven (pred (pred x))) 2 </pre>
---	---	---

This, in turn, allows us to apply rule E-FIX1. The symbol `iseven` is replaced with the recursive function:

<pre> fix (λ iseven : Nat → Bool. λ x : Nat. if iszero x then true else if iszero (pred x) then false else iseven (pred (pred x))) 2 </pre>	→	<pre> (λ x : Nat. if iszero x then true else if iszero (pred x) then false else fix(...)(pred (pred x))) 2 </pre>
---	---	---

Next, a standard function application via rule E-APPABS leads to the following term:

<pre> (λ x : Nat. if iszero x then true else if iszero (pred x) then false else fix(...)(pred (pred x))) 2 </pre>	→	<pre> if iszero 2 then true else if iszero (pred 2) then false else fix(...)(pred (pred 2)) </pre>
---	---	--

As neither `iszero 2` nor `iszero (pred 2)` evaluate to true, repeated application of the evaluation rules lead to the following term:

<pre> if iszero 2 then true else if iszero (pred 2) then false else fix(...)(pred (pred 2)) </pre>	→*	<pre> fix (λ iseven : Nat → Bool. λ x : Nat. if iszero x then true else if iszero (pred x) then false else iseven (pred (pred x))) (pred (pred 2)) </pre>
--	----	---

Again, rule E-FIX1 substitutes the symbol `iseven` for the actual definition of the function. Furthermore, evaluating the term `pred (pred 2)` by applying rule E-PRESUCC two times leads to the following term:

<pre> fix (λ iseven : Nat → Bool. λ x : Nat. if iszero x then true else if iszero (pred x) then false else iseven (pred (pred x))) (pred (pred 2)) </pre>	→*	<pre> (λ x : Nat. if iszero x then true else if iszero (pred x) then false else fix(...)(pred (pred x))) 0 </pre>
---	----	---

Applying the function via rule E-APPABS substitutes x for 0 . The guard of the first if-then-else expression `iszero 0` then evaluates to true, leading to the final term `true`:

```
( $\lambda x : \text{Nat.}$ 
  if iszero x then true
  else if iszero (pred x) then false
  else fix(...)(pred (pred x)) 0)  $\longrightarrow^*$  true
```

3.2.2. Records

λ_{Records}	Extends $\lambda_{\text{Recursion}}$ (Figure 3.4)
<p><i>New syntactic elements</i></p> <p>$t ::= \dots$ <i>terms:</i> $\{l_i = t_i^{i \in 1 \dots n}\}$ record $t.l$ record projection</p> <p>$v ::= \dots$ <i>values:</i> $\{l_i = v_i^{i \in 1 \dots n}\}$ record value</p> <p>$T ::= \dots$ <i>types:</i> $\{l_i : T_i^{i \in 1 \dots n}\}$ record type</p> <p><i>New evaluation rules</i> $t \longrightarrow t'$</p> <p>$\{l_i = v_i^{i \in 1 \dots n}\}.l_j \longrightarrow v_j$ (E-PROJRCD)</p> <p>$\frac{t_1 \longrightarrow t'_1}{t_1.l \longrightarrow t'_1.l}$ (E-PROJ)</p> <p>$t_j \longrightarrow t'_j$</p> <hr/> <p>$\{l_i = v_i^{i \in 1 \dots j-1}, l_j = t_j, l_k = t_k^{k \in j+1 \dots n}\} \longrightarrow$ $\{l_i = v_i^{i \in 1 \dots j-1}, l_j = t'_j, l_k = t_k^{k \in j+1 \dots n}\}$ (E-RCD)</p>	<p><i>New typing rules</i> $\Gamma \vdash t : T$</p> <p>$\frac{\text{for each } i \quad \Gamma \vdash t_i : T_i}{\Gamma \vdash \{l_i = t_i^{i \in 1 \dots n}\} : \{l_i : T_i^{i \in 1 \dots n}\}}$ (T-RCD)</p> <p>$\frac{\Gamma \vdash t_1 : \{l_i : T_i^{i \in 1 \dots n}\}}{\Gamma \vdash t_1.l_j : T_j}$ (T-PROJ)</p> <p><i>New subtyping rules</i> $T <: T'$</p> <p>$\{l_i : T_i^{i \in 1 \dots n+k}\} <: \{l_i : T_i^{i \in 1 \dots n}\}$ (S-RCDWIDTH)</p> <p>$\frac{\text{for each } i \quad T_i <: T'_i}{\{l_i : T_i^{i \in 1 \dots n}\} <: \{l_i : T'_i^{i \in 1 \dots n}\}}$ (S-RCDDEPTH)</p> <p>$\frac{\{k_j : T_j^{j \in 1 \dots n}\} \text{ is a permutation of } \{l_i : T'_i^{i \in 1 \dots n}\}}{\{k_j : T_j^{j \in 1 \dots n}\} <: \{l_i : T'_i^{i \in 1 \dots n}\}}$ (S-RCDPERM)</p>

Figure 3.5.: Rules for a λ -calculus enriched with records.

Records are compound data structures for grouping data based on labels. A record $\{l_i = t_i^{i \in 1 \dots n}\}$

can be understood as a n-ary tuple in which each field t_i is annotated by a label l_i [81]. Likewise, a record type $\{l_i : T_i \mid i \in 1 \dots n\}$ consists of labels l_i and types T_i that represent the types of the terms in the record $\{l_i = t_i \mid i \in 1 \dots n\}$. We generally assume that all labels in a record are unique and that the order in which label-value pairs are given is irrelevant. As examples, consider the record values $\{\text{age} = 20\}$ and $\{\text{matrNr} = 211023, \text{enrolled} = \text{true}\}$ with their respective types $\{\text{age} : \text{Nat}\}$ and $\{\text{matrNr} : \text{Nat}, \text{enrolled} : \text{Bool}\}$. Terms in records are accessed via a projection operation $t.l$ that returns the term that is annotated with the label, for example $\{\text{matrNr} = 211023, \text{enrolled} = \text{true}\}.\text{matrNr}$ for accessing the matriculation number label of the record. Figure 3.5 summarizes the rules regarding records.

A record is considered to be a value if all its terms are values. If not, a term of the record that is not yet a value is evaluated (rule E-RCD). In case of a projection $t.l$, the term t is reduced until it is a value (rule E-PROJ). If t is a record value, then the value annotated with the label l is returned (rule E-PROJRCR). Typing a record essentially means typing each term of the record individually (rule T-RCD). Typing a projection $t_1.l$ requires t_1 be a record type that contains the label l . If so, then the type associated to the label l can be the type of the term (rule T-PROJ). Lastly, subtyping between record types is defined through the annotated types. A record type is more specific than another record type if it contains more labels and the types of all labels occurring in both records are in subtype relations (rule S-RCDWIDTH). For example, the record $\{a : \text{Nat}, b : \text{Bool}\}$ is more specific than $\{a : \text{Nat}\}$ and therefore the former is a subtype of the latter. A record is also more specific than another record if it contains the same labels and their types are in a subtype relation (rule S-RCDDEPTH). Lastly, rule S-RCDPERM ensures that the order of label-type pairs is irrelevant for the subtype relation.

Multiple subtyping rules can be combined through rule S-TRANS. As an example, consider the derivation of the type $\{b : \text{Top}\}$ for the term $\{a : 42, b : \text{true}\}$. The term is of type $\{a : \text{Nat}, b : \text{Bool}\}$ is a subtype of $\{b : \text{Top}\}$:

$$\frac{\frac{}{\{a : \text{Nat}, b : \text{Bool}\} <: \{b : \text{Bool}\}} \text{S-RCDWIDTH} \quad \frac{\frac{}{\text{Bool} <: \text{Top}} \text{S-TOP} \quad \frac{}{\{b : \text{Bool}\} <: \{b : \text{Top}\}} \text{S-RCDDEPTH}}{\{a : \text{Nat}, b : \text{Bool}\} <: \{b : \text{Top}\}} \text{S-TRANS}}$$

The complete derivation of the statement $\{a : 42, b : \text{true}\} : \{b : \text{Top}\}$ is therefore as follows:

$$\frac{\frac{\vdots \text{ T-RCD} \quad \vdots}{\emptyset \vdash \{a : 42, b : \text{true}\} : \{a : \text{Nat}, b : \text{Bool}\}} \quad \frac{}{\{a : \text{Nat}, b : \text{Bool}\} <: \{b : \text{Top}\}} \text{S-TRANS}}{\emptyset \vdash \{a : 42, b : \text{true}\} : \{b : \text{Top}\}} \text{ T-SUB}$$

Algorithmic type checking Again, the type checking function `TYPEOF` as shown in Algorithm 4 introduces one new case per rule of the type system (see Figure 3.5). The implementation

diverges from the rules of the type system. Implementations of record subtyping commonly rely on a subtyping rule that combines the rules S-RCDWIDTH, S-RCDDEPTH and S-RCDPERM (c. f. [81]). Subtyping first compares the labels of the record—all labels of the supertype must occur in the subtype. Second, for each label that occurs in both records, their types must be in a subtype relation:

$$\frac{\{l_j^{j \in 1 \dots m}\} \subseteq \{l_i^{i \in 1 \dots n}\} \quad l_i = l_j \text{ implies } T_i <: T_j}{\{l_i : T_i^{i \in 1 \dots n}\} <: \{l_j : T_j^{j \in 1 \dots m}\}} \quad (\text{S-RCD})$$

Using this rule, the procedures TYPEOF and SUBTYPE can then be implemented as shown in Algorithm 4. Records become more specific the more labels they contain. As such, the least upper bound (LUB) of

Algorithm 4 Definition of TYPEOF and SUBTYPE for $\lambda_{Records}$.

```

function TYPEOF( $t, \Gamma$ )
  match  $t$  with
    case ... then ...       $\triangleright$  existing cases
    case  $\{l_i = t_i^{i \in 1 \dots n}\}$  when for all  $i \in \{1 \dots n\}$ : TYPEOF( $t_i, \Gamma$ ) is  $T_i$  then  $\{l_i : T_i^{i \in 1 \dots n}\}$ 
    case  $t_1.l_j$  when TYPEOF( $t_1, \Gamma$ ) is  $\{l_i : T_i^{i \in 1 \dots n}\}$  and  $l_j \in \{l_i^{i \in 1 \dots n}\}$  then  $T_j$ 
    case _ then FAIL

function SUBTYPE( $T, T'$ )
  match ( $T, T'$ ) with
    case ... then ...       $\triangleright$  existing cases
    case ( $\{t_i : T_i^{i \in 1 \dots n}\}, \{t_j : T_j^{j \in 1 \dots m}\}$ ) when  $\{l_j^{j \in 1 \dots m}\} \subseteq \{l_i^{i \in 1 \dots n}\}$ 
      and for all  $i \in \{1 \dots n\}$ : there is some  $j \in 1, \dots, m$ 
        with  $l_i = l_j$  and SUBTYPE( $T_i, T_j$ ) then
          true
    case _ then false

```

two records is the record that only contains labels that occur in both records. Likewise, the greatest lower bound (GLB) must combine labels of both records. Algorithm 5 contains the definitions for both functions.

3.2.3. Lists

Lastly, we introduce lists as a second compound data structure into the programming language. The type `List T` represents a finite length lists of values that belong to type T . A list is either the empty list, represented by the term `nil[T]`, or a pair, represented by the term `cons $t_1 t_2$` , which contains a term followed by a list. A list containing the numbers 0 to 2 is represented as follows as

Algorithm 5 Definition of LUB and GLB for $\lambda_{Records}$.

```

function LUB( $T, T'$ )
  match ( $T, T'$ ) with
    case ... then ...      ▷ existing cases
    case ( $\{l_i : T_i^{i \in 1 \dots n}\}, \{l_j : T_j^{j \in 1 \dots m}\}$ ) when for all  $l_k \in \{l_i\} \cap \{l_j\}$  lub( $T_i, T_j$ ) is  $T_k$  then
       $\{l_k : T_k\}$ 
    case _ then Top

function GLB( $T, T'$ )
  match ( $T, T'$ ) with
    case ... then ...      ▷ existing cases
    case ( $\{l_i : T_i^{i \in 1 \dots n}\}, \{l_j : T_j^{j \in 1 \dots m}\}$ ) when for all  $l_k \in \{l_i\} \cup \{l_j\}$  glb( $T_i, T_j$ ) is  $T_k$  then
       $\{l_k : T_k\}$ 
    case _ then FAIL
  
```

`cons 0 (cons 1 (cons 2 nil[Nat]))` . Compared to other values, `nil` is different as it carries a type annotation. As `nil` occurs in all types of lists, this is required to distinguish empty lists. For example, an empty list of type `Nat` is represented as `nil[Nat]` whereas an empty list of boolean values is `nil[Bool]` . Furthermore, several standard constructs for the usage of lists such as `head` and `tail` as well as `isNil` for checking whether a list is empty are added to the language. Figure 3.6 summarizes the rules regarding lists. Terms are reduced until they are values (rules E-HEAD2, E-TAIL2 and E-ISNIL3). Once terms have been reduced to values, `head` and `tail` can be evaluated by taking either the first or second term associated with the `cons` term (rules E-HEAD1 and E-TAIL1). Evaluating `isNil` to either true or false is done depending on whether the term is a `cons` term or `nil` (rules E-ISNIL1 and E-ISNIL2). Typing rules are straightforward. In case of `nil[T1]`, the type annotation is used to construct the type `List T1` (rule T-NIL). In case of `cons` , it is checked whether both t_1 and t_2 are of type T_1 (rule T-CONS). `isNil` , `head` and `tail` all expect lists of type T_1 and return either a value of type `Bool` (rule T-ISNIL), a value of type T_1 (rule T-HEAD) or a list of type T_1 (rule T-TAIL). In terms of subtyping, a list `List T` is a subtype of `List T'` if T is a subtype of T' since then all values of T are also values of T' .

To highlight the language so far, we can define a *map* function—a higher order function that takes a function and applies it on all values of a list and returns the resulting list. While we cannot formulate a generic version of *map* due to the lack of parametric polymorphism, we can define a basic version. For example, we can define a version of the map-function `mapNatBool` that can be used for all functions of the type `Nat → Bool` . We can then use the previously defined λ -abstraction `iseven` to map a list of

New syntactic elements

$t ::=$...	<i>terms:</i>
$\mathbf{nil}[T]$		constant empty list
$\mathbf{cons} \ t \ t$		list constructor
$\mathbf{isNil} \ t$		test for empty list
$\mathbf{head} \ t$		head of a list
$\mathbf{tail} \ t$		tail of a list

$v ::=$...	<i>values:</i>
$\mathbf{nil}[T]$		empty list
$\mathbf{cons} \ v \ v$		list value

$T ::=$...	<i>types:</i>
$\mathbf{List} \ T$		type of lists

New evaluation rules

	$t \longrightarrow t'$	
$\mathbf{head} \ (\mathbf{cons} \ v_1 \ v_2) \longrightarrow v_1$	(E-HEAD1)	
$\frac{t_1 \longrightarrow t'_1}{\mathbf{head} \ t_1 \longrightarrow \mathbf{head} \ t'_1}$	(E-HEAD2)	
$\mathbf{tail} \ (\mathbf{cons} \ v_1 \ v_2) \longrightarrow v_2$	(E-TAIL1)	
$\frac{t_1 \longrightarrow t'_1}{\mathbf{tail} \ t_1 \longrightarrow \mathbf{tail} \ t'_1}$	(E-TAIL2)	
$\frac{t_1 \longrightarrow t'_1}{\mathbf{cons} \ t_1 \ t_2 \longrightarrow \mathbf{cons} \ t'_1 \ t_2}$	(E-CONS1)	
$\frac{t_2 \longrightarrow t'_2}{\mathbf{cons} \ v_1 \ t_2 \longrightarrow \mathbf{cons} \ v_1 \ t'_2}$	(E-CONS2)	

$$\mathbf{isNil} \ (\mathbf{nil}[T]) \longrightarrow \mathbf{true} \quad (\text{E-ISNIL1})$$

$$\mathbf{isNil} \ (\mathbf{cons} \ t_1 \ t_2) \longrightarrow \mathbf{false} \quad (\text{E-ISNIL2})$$

$$\frac{t_1 \longrightarrow t'_1}{\mathbf{isNil} \ t_1 \longrightarrow \mathbf{isNil} \ t'_1} \quad (\text{E-ISNIL3})$$

New typing rules

$$\boxed{\Gamma \vdash t : T}$$

$$\Gamma \vdash \mathbf{nil}[T_1] : \mathbf{List} \ T_1 \quad (\text{T-NIL})$$

$$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash \mathbf{cons} \ t_1 \ t_2 : \mathbf{List} \ T_1} \quad (\text{T-CONS})$$

$$\frac{\Gamma \vdash t_1 : \mathbf{List} \ T_1}{\Gamma \vdash \mathbf{isNil} \ t_1 : \mathbf{Bool}} \quad (\text{T-ISNIL})$$

$$\frac{\Gamma \vdash t_1 : \mathbf{List} \ T_1}{\Gamma \vdash \mathbf{head} \ t_1 : T_1} \quad (\text{T-HEAD})$$

$$\frac{\Gamma \vdash t_1 : \mathbf{List} \ T_1}{\Gamma \vdash \mathbf{tail} \ t_1 : \mathbf{List} \ T_1} \quad (\text{T-TAIL})$$

New subtyping rules

$$\boxed{T <: T'}$$

$$\frac{T <: T'}{\mathbf{List} \ T <: \mathbf{List} \ T'} \quad (\text{S-LIST})$$

Figure 3.6.: Rules for lists in the λ -calculus.

numerical values to a list containing either `true` or `false` depending on whether the numerical value is even or odd:

```

1  letrec mapNatBool : (Nat → Bool) → (Nat list → Bool list) =
2    λ(f:Nat → Bool) . λ(l:Nat list) .
3      if (isNil l) then nil[Bool]
4      else cons (f (head l)) (mapEvenOdd f (tail l)) in
5  letrec iseven : Nat → Bool = λ(x:Nat) .
6    if iszero x then true
7    else if iszero (pred x) then false
8    else iseven (pred (pred x))
9  mapNatBool iseven (cons 0 (cons 1 (cons 2 nil[Nat])))

```

▷ as defined in Subsection 3.2.1

A noteworthy limitation that is introduced by the addition of lists is that lists can be the source of run-time errors that cannot be detected with a type-system as described in this thesis. Consider the term `head nil[Nat]`. This term is well-typed:

$$\frac{\frac{}{\emptyset \vdash \text{nil}[\text{Nat}] : \text{List Nat}}{\text{T-NIL}}}{\emptyset \vdash \text{head nil}[\text{Nat}] : \text{Nat}} \text{T-HEAD}$$

However, the term clearly is not a value, but also cannot be reduced any further with the evaluation rules given in Figure 3.6. Therefore, it is common to formulate type-safety such that a well-typed term does not get stuck unless the program reaches a point where it tries to compute `head nil[T]` or `tail nil[T]`.

Algorithmic type checking Algorithmic type checking is a straightforward implementation of the rules in Figure 3.6. Both, `TYPEOF` and `SUBTYPE` are extended with new cases (see Algorithm 6). Likewise, construction of the least upper bound (LUB) and greatest lower bound (GLB) of two lists `List T` and `List T'` simply uses the least upper bound or greatest lower bound of `T` and `T'` (see Algorithm 7).

Algorithm 6 Definition of `TYPEOF` and `SUBTYPE` for λ_{Full} .

```
function TYPEOF( $t, \Gamma$ )  
  match  $t$  with  
    case ... then ...       $\triangleright$  existing cases  
    case nil[ $T$ ] then  $T$   
    case cons  $t_1 t_2$  when TYPEOF( $t_1, \Gamma$ ) is  $T_1$  and TYPEOF( $t_2, \Gamma$ ) is List  $T_2$  and  
      LUB( $T_1, T_2$ ) IS List  $T_{lub}$  then  $T_{lub}$   
    case isNil  $t_1$  when TYPEOF( $t_1, \Gamma$ ) is List  $T_1$  then Bool  
    case head  $t_1$  when TYPEOF( $t_1, \Gamma$ ) is List  $T_1$  then  $T_1$   
    case tail  $t_1$  when TYPEOF( $t_1, \Gamma$ ) is List  $T_1$  then List  $T_1$   
    case _ then FAIL  
  
function SUBTYPE( $T, T'$ )  
  match ( $T, T'$ ) with  
    case ... then ...       $\triangleright$  existing cases  
    case (List  $T, \text{List } T'$ ) when SUBTYPE( $T, T'$ ) then true  
    case _ then false
```

Algorithm 7 Definition of `LUB` and `GLB` for λ_{Full} .

```
function LUB( $T, T'$ )  
  match ( $T, T'$ ) with  
    case ... then ...       $\triangleright$  existing cases  
    case (List  $T, \text{List } T'$ ) then List LUB( $T, T'$ )  
    case _ then Top  
  
function GLB( $T, T'$ )  
  match ( $T, T'$ ) with  
    case ... then ...       $\triangleright$  existing cases  
    case (List  $T, \text{List } T'$ ) then List GLB( $T, T'$ )  
    case _ then FAIL
```

Type Checking with Description Logics

Developing applications that work on RDF data is error prone since problems, such as accessing properties which may not exist, only surface at run-time. Ontologies, which are rooted in description logics, provide some form of schematic description for the data. As such, they can be used for type checking. We therefore introduce λ_{DL} , a research language that features type-safe programming using a description logic knowledge base both during type checking as well as during run-time. This chapter is based on [65].

4.1. Key Design Principles and Example Use Case

Key Design Principles The language λ_{DL} is based on several key design principles:

Programs are defined with respect to a knowledge base. All programs are defined with respect to a knowledge base that provides both, the data the program works on as well as schematic information about the data. Subsequently, queries in the program are evaluated using the knowledge base. Likewise, type checking of programs is defined using the knowledge base.

Concepts as types. As introduced in Section 2.1.2, types represent sets of values. The integration of semantic data into a programming language therefore requires types that can represent sets of graph nodes. Concept expressions as introduced in Section 2.2.3 provide highly expressive descriptions of sets of graph nodes and therefore constitute a new form of type that is to be integrated into the programming language.

Subtype entailment. Integration of concept expressions as types calls for a subtype relation between this new form of types. Subtyping between concept expressions cannot be decided purely based on syntax. Furthermore, there are also infinitely many syntactic variations of any given concept, making it impossible to precompute all possible subtype relations. Instead of somehow mapping the concept inclusion rules into rules for a type system and thereby replicating the reasoning

process part of the knowledge base, we rather forward the subtyping decision to the knowledge base.

Typing of queries. Queries constitute the main form of data access. As such, they must be checked in two ways: First, unsatisfiable queries—that is queries for which it is impossible to produce a result—must be rejected. Second, queries must be assigned meaningful types which represent the graph nodes that the query evaluates to.

Open-world querying. Looking at logical entailment, any statement is either *true*, *false* or *unknown*. A type system must only consider certain knowledge as true—that is, we only consider statements to be true which are true in all models of the knowledge base. In all other cases, we consider the statement to be false. While this is close to a developer’s expectation, it creates an interesting side effect. Given any concept expression C , asking whether an arbitrary graph node is an instance of $C \sqcup \neg C$ will always be true. However, first asking whether a node is an instance of C then separately asking whether it is an instance of $\neg C$ may be false in both cases.

Example Use Case As an example, consider an application that is defined with respect to the following knowledge base as shown in Figure 4.1: A `Person` has a name (line 1) while a `University` has a location (line 2). If someone studies at a `University`, then it must be a `Student` (line 3). A `Student` is a `Person` and all of his or her friends are also instances of `Student` (line 4). A `Professor` is a `Person` who works at a `University` (line 5). In terms of data, there is `uniKo` which is an instance of `University` (line 7). `bob` studies at `uniKo` (line 8), making him an instance of `Student`. Furthermore, his name is “Bob B.” (line 9) and he is friends with `alice` (line 10). As `bob` is a `Student` and all of a student’s friends are also `Students`, it means that `alice` must also be a `Student`. The name of `alice` is “Alice A.” (line 11). Lastly, `charlie` is an instance of `Professor` (line 12).

Our example application should implement three different functions: First, it should query for all instances of `Student`. Second, it should define a function that is able to return the name of a given `Person`. Third, it should define a function that returns the list of friends for a given `Student`.

Querying for all instances of `Student` is straightforward as queries are directly integrated into the language:

```
1 query x ← x type Student
```

The expression will evaluate to a list of records where each record has a single label representing the projection variable `x`. The type of the record label is a type that is equivalent to the concept expression `Student`. A function that returns a name of a `Person` can be defined as follows:

```
1 let getName = λ (p:Person) . p.hasName
```

```

1  Person  $\sqsubseteq$   $\exists$  hasName.  $\top$ 
2  University  $\sqsubseteq$   $\exists$  location.  $\top$ 
3   $\exists$  studiesAt. University  $\sqsubseteq$  Student
4  Student  $\sqsubseteq$  Person  $\sqcap$   $\forall$  hasFriend. Student
5  Professor  $\sqsubseteq$  Person  $\sqcap$   $\exists$  worksAt. University
6
7  uniKo : University
8  (bob, uniKo) : studiesAt
9  (bob, "Bob B.") : hasName
10 (bob, alice) : hasFriend
11 (alice, "Alice A.") : hasName
12 charlie : Professor

```

Figure 4.1.: Example knowledge base K_4 .

The input type of the function is the concept expression `Person`. In the body of the function, a role projection can be used to traverse the graph from the given graph node to all nodes reachable via the `-hasName→` relation. The result of the function will be a list of graph nodes belonging to the concept expression `\exists hasName-.Person`, essentially expressing that the graph node was reached by traversing the `-hasName→` relation from a `Person`. Subsequently, the function is of type `getName : Person → List \exists hasName-.Person`. The function can for example be applied to instances of the concept expression `Student` as the `Person` concept expression subsumes the `Student` concept expression. The function can therefore be applied to the query results of the query above. Lastly, a function that returns all friends of a `Student` introduces casting. This is because it is only known that all friends of a `Student` are `Students` again—not that a `Student` actually has a friend. λ_{DL} provides a type-safe form of casting through the typecase statement that allows to proceed on a case by case basis:

```

1  let getFriends =  $\lambda$  (s:Student) .
2    case s of
3      type  $\exists$  hasFriend.  $\top$  as y → y.hasFriend
4      default nil[Student]

```

Using the knowledge base, the expression first checks if the given `Student` is an instance of `\exists hasFriend. \top` . If this is the case, then it is safe to use a role projection to access the list of friends. Since it is known that an instance of `Student` can only have other instances of `Student` as a friend, the resulting list is equivalent to `List Student`. If it is not possible to show that the object is an instance

of $\exists \text{ hasFriend.T}$, the expression resorts to the **default** case which returns an empty list. The function is of type $\text{getFriends} : \text{Student} \rightarrow \text{List Student}$.

4.2. Types for Conjunctive Queries

To provide a typed integration of conjunctive queries in a programming language, we must reconstruct types based on the query. To support the assignment of types in the type system, we first reconstruct a set of axioms from the query. Queries evaluate to sets of mappings which map variables onto graph nodes. We represent each variable through an atomic concept and then note all the constraints on this concept. That is, we assume the possibility to transform each variable x of a query into an atomic concept A_x which must be unique for the variable as well as the query. Variables which share a name but occur in different queries should therefore be represented by different concepts. We then examine each graph pattern (abbreviated as gp) of the query $\bar{x} \leftarrow gp$ and impose constraints on these atomic concepts based on the graph pattern. These constraints are formulated as concept expressions C . The result of this process is a set of axioms K_q that take the form $A_x \sqsubseteq C$.

Typing rules for SPARQL CQs		$q : K_q$
$x \text{ type concept} : \{A_x \sqsubseteq A_{\text{concept}}\}$	(QT-TYPE)	
$\frac{r \neq \text{type}}{x \text{ r o} : \{A_x \sqsubseteq \exists r.\{o\}\}}$	(QT-ROLE1)	$\frac{r \neq \text{type}}{x_1 \text{ r } x_2 : \{A_{x_1} \sqsubseteq \exists r.A_{x_2}, A_{x_2} \sqsubseteq \exists r^-.A_{x_1}\}}$ (QT-ROLE3)
$\frac{r \neq \text{type}}{o \text{ r x} : \{A_x \sqsubseteq \exists r^-. \{o\}\}}$	(QT-ROLE2)	$\frac{gp_1 : K_{q_1} \quad gp_2 : K_{q_2}}{gp_1 \wedge gp_2 : K_{q_1} \cup K_{q_2}}$ (QT-CONJ)
		$\frac{gp : K_q}{\bar{x} \leftarrow gp : K_q}$ (QT-PROJ)

Figure 4.2.: Rules for assigning concepts to variables in queries.

Figure 4.2 summarizes the rules for inferring axioms from queries. In case of graph patterns, we distinguish between queries that are asking for instances of a specific concept and other graph patterns. If a query asks for the former, then we simply map the variable onto the concept (rule QT-TYPE). In other cases, we construct existentially quantified concept expressions using the given role r that point to nominal concepts $\exists r.\{o\}$ or $\exists r^-. \{o\}$ (rule QT-ROLE1 and QT-ROLE2). If variables occur in both subject and object position of the graph pattern, we construct existentially quantified concepts that use the atomic concept names representing the respective variable (rule QT-ROLE3). Conjunction of graph patterns (abbreviated as gp) infers a set of axioms K_{q_1} and K_{q_2} for gp_1 and gp_2 individually and then

takes the union of the two sets (rule QT-CONJ). Any interpretation that satisfies the resulting set has to satisfy each axiom in the set. If both sets contain constraints about the same atomic concept, this is equivalent to taking the conjunction of the two constraints. For example, if $K_{q_1} = \{A_x \sqsubseteq \text{Student}\}$ and $K_{q_2} = \{A_x \sqsubseteq \text{Person}\}$, then $K_{q_1} \cup K_{q_2} = \{A_x \sqsubseteq \text{Student}, A_x \sqsubseteq \text{Person}\}$ is semantically equivalent to $\{A_x \sqsubseteq \text{Student} \sqcap \text{Person}\}$. Projection does not have any effect on typing (rule QT-PROJ).

As an example consider the query:

$$q_1 = x \leftarrow x \text{ type Student} \wedge x \text{ studiesAt } y$$

The query contains the two graph patterns $gp_1 = x \text{ type Student}$ and $gp_2 = x \text{ studiesAt } y$. Both graph patterns are examined individually (rule QT-CONJ). In case of gp_1 , rule QT-TYPE applies. That is, the pattern $x \text{ type Student}$ is assigned the set of axioms $\{A_x \sqsubseteq \text{Student}\}$ where A_x refers to the atomic concept representing the variable x (rule QT-TYPE). Second, the graph pattern $x \text{ studiesAt } y$ is assigned to the set of axioms $\{A_x \sqsubseteq \exists \text{studiesAt}.A_y, A_y \sqsubseteq \exists \text{studiesAt}^-.A_x\}$ via rule QT-ROLE3. Rule QT-CONJ then takes the union of both sets of axioms, yielding the following set of axioms:

$$q_1 : K_{q_1} = \{ A_x \sqsubseteq \text{Student}, \\ A_x \sqsubseteq \exists \text{studiesAt}.A_y, \\ A_y \sqsubseteq \exists \text{studiesAt}^-.A_x \}$$

4.3. Core Language

Syntax and Semantics The language λ_{DL} (see Figure 4.3) is an extension of the standard call-by-value λ -calculus as presented in Chapter 3. New terms (denoted by t) include the **query** keyword for issuing queries and projection from an object to a set of objects via a role expression r . As an example for a simple projection, consider the term bob.studiesAt essentially meaning a traversal from the node (bob) to all nodes reachable via the $-\text{studiesAt} \rightarrow$ relation. Furthermore, we introduce an operator for equality between values. Values (denoted by v) now also include graph nodes o . A new form of types (denoted by T) are concept expressions that are built according to Figure 2.7.

The operational semantics bears no significant differences to the standard ones as defined in Chapter 3. Evaluation is now defined with respect to a knowledge base K . However, this has no impact on the standard reduction rules that are unrelated to newly added constructs. We therefore do not explicitly repeat them, but rather focus on the new evaluation rules. Evaluation of a projection can be rewritten into a query (rule E-PROJROLE). If the projection term is not yet a value and can take a step, it will be reduced (rule E-RPROJ). Equality of graph nodes relies on the given knowledge base

K . Even though they are syntactically different, they may be semantically equivalent¹. Therefore, the knowledge base K has to either show that two graph nodes are semantically equivalent (rule E -EQ-NODE) or they are assumed to be different (rule E -NEQ-NODE). That means, we treat not knowing whether they are semantically equivalent and knowing they are not in the same manner. Equality of other primitive values works by comparing syntactic equality (rules E -EQ-PRIM and E -NEQ-PRIM). In case the two terms of the equality are not yet values, they are evaluated successively (rules E -EQ1 and E -EQ2). Querying data via the **query**-keyword evaluates the query q over the knowledge base K . This results in a set of mappings μ . These mappings are then converted into a list of records by taking each projection variable and turning it into a label of a record. The value referenced by the record label is then the graph node to which a given μ maps the variable (rule E -QUERY). We chose to convert query results into a list of records rather than a set even though this creates an implicit ordering. However, lists are a more basic programming language construct and subsequent processing of query results introduces an ordering anyways.

Type System and Subtyping The most distinguishing feature of the type system (see Figure 4.3) is the addition of concept expressions c as well as the type reconstruction on queries. Besides the usual context Γ that is present in type checking, we now also require a knowledge base K to make judgments during the type checking process. For normal constructs unrelated to the knowledge base, this has little impact. We therefore again omit typing rules that are unrelated to the knowledge base and focus on the newly introduced syntactic expressions. A projection term $t_1.r$ can be assigned to the type $\text{List } \exists r^-.C_1$ if two conditions hold: For one, it must be possible to assign t_1 to the type concept expression C_1 . If this is the case, then it is known that t_1 evaluates to a graph node. For two, the concept expression C_1 must be subsumed by $\exists r.\top$, indicating that a relation r exists for the objects of the concept expression C_1 . Typing equality $t_1 = t_2$ simply requires that both t_1 and t_2 are either typed with concept expressions (rule T -EQ-NOM) or primitive types (rule T -EQ-PRIM). The type of the complete expression is then a `Bool`. Lastly, terms should always be typed with their principal type—that is, the most specific type possible. For a graph nodes o , we therefore resort to the nominal concept expressions that only contains this graph node.

In case of queries (rule T -QUERY), we use rules given in Figure 4.2 to retrieve the set of axioms K_q containing the information about the atomic concepts A_x for each variable used in the query q . We then proceed to check each variable for satisfiability—in case there is a variable for which $K \cup K_q \models A_x \sqsubseteq \perp$, then this variable is not satisfiable. If a query contains an unsatisfiable query variable, there cannot be any answers to the query. We subsequently do not assign a type and therefore reject the query. If all variables are satisfiable, we proceed to construct a record type $\{l_i : A_{l_i}\}$

¹This may e.g. occur when combining different data sets—the data sets may use different identifier that semantically refer to the same object. They can be merged by adding equivalence statements for such identifiers.

λ_{DL} Extends λ_{Full} (Fig. 3.6)

New syntactic elements

$t ::= \dots$ *terms:*
 | **query** q SPARQL CQ
 | $t.r$ role projection
 | $t = t$ equivalence

$v ::= \dots$ *values:*
 | o graph node

$T ::= \dots$ *types:*
 | C concept expression

New evaluation rules

$$\boxed{t \xrightarrow{K} t'}$$

$$\frac{\llbracket x_1 \leftarrow o_1 r x_1 \rrbracket_K = \{\mu_i^{i \in 1 \dots n}\}}{o_1.r \xrightarrow{K} \mathbf{cons} \mu_1(x_1) \dots \mathbf{cons} \mu_n(x_1) \mathbf{nil}} \quad (\text{E-PROJROLE})$$

$$\frac{t_1 \xrightarrow{K} t'_1}{t_1.r \xrightarrow{K} t'_1.r} \quad (\text{E-RPROJ})$$

$$\frac{K \models o_1 \equiv o_2}{o_1 = o_2 \xrightarrow{K} \mathbf{true}} \quad (\text{E-EQ-NODE})$$

$$\frac{K \not\models o_1 \equiv o_2}{o_1 = o_2 \xrightarrow{K} \mathbf{false}} \quad (\text{E-NEQ-NODE})$$

$$\pi_1 = \pi_1 \xrightarrow{K} \mathbf{true} \quad (\text{E-EQ-PRIM})$$

$$\frac{\pi_1 \text{ and } \pi_2 \text{ syntactically different}}{\pi_1 = \pi_2 \xrightarrow{K} \mathbf{false}} \quad (\text{E-NEQ-PRIM})$$

$$\frac{t_1 \xrightarrow{K} t'_1}{t_1 = t_2 \xrightarrow{K} t'_1 = t_2} \quad (\text{E-EQ1})$$

$$\frac{t_2 \xrightarrow{K} t'_2}{v_1 = t_2 \xrightarrow{K} v_1 = t'_2} \quad (\text{E-EQ2})$$

$$\frac{\llbracket q \rrbracket_K = \{\mu_i^{i \in 1 \dots n}\} \quad \text{head}(q) = \{l_j^{j \in 1 \dots m}\}}{\mathbf{query} \ q \xrightarrow{K} \mathbf{cons} \{l_j = \mu_1(l_j)^{j \in 1 \dots m}\} \dots \mathbf{cons} \{l_j = \mu_n(l_j)^{j \in 1 \dots m}\} \mathbf{nil}} \quad (\text{E-QUERY})$$

New typing rules

$$\boxed{\Gamma, K \vdash t : T}$$

$$\frac{\Gamma, K \vdash t_1 : C_1 \quad K \models C_1 \sqsubseteq \exists r. T}{\Gamma, K \vdash t_1.r : \text{List}(\exists r. C_1)} \quad (\text{T-RPROJ})$$

$$\frac{\Gamma, K \vdash t_1 : C \quad \Gamma, K \vdash t_2 : D}{\Gamma, K \vdash t_1 = t_2 : \text{Bool}} \quad (\text{T-EQ-NOM})$$

$$\frac{\Gamma, K \vdash t_1 : \Pi_1 \quad \Gamma, K \vdash t_2 : \Pi_1}{\Gamma, K \vdash t_1 = t_2 : \text{Bool}} \quad (\text{T-EQ-PRIM})$$

$$\Gamma, K \vdash o : \{o\} \quad (\text{T-NOMINAL})$$

$$\frac{q : K_q \quad \text{head}(q) = \{l_i^{i \in 1 \dots m}\} \quad \forall x \in \text{Vars}(q) : K \cup K_q \not\models A_x \sqsubseteq \perp}{\Gamma, K \cup K_q \vdash \mathbf{query} \ q : \{l_i : A_i^{i \in 1 \dots m}\} \text{list}} \quad (\text{T-QUERY})$$

$$\frac{\Gamma, K \cup \{A_i \sqsubseteq C_i^{i \in 1 \dots n}\} \vdash t : A_j^{1 \leq j \leq n} \quad K \cup \{A_i \sqsubseteq C_i^{i \in 1 \dots n}\} \models A_j \sqsubseteq D^{1 \leq j \leq n}}{\Gamma, K \vdash t : D} \quad (\text{T-ADD})$$

New subtyping rules

$$\boxed{K \vdash T <: T'}$$

$$\frac{K \models C \sqsubseteq D}{K \vdash C <: D} \quad (\text{S-CONCEPT})$$

Figure 4.3.: Syntax, Semantics and type system rules for λ_{DL} .

where $l_i^{i \in 1 \dots m}$ represent the projection variables of the query. The final type that is assigned to the query term using $K \cup K_q$ is the list of records $\{l_i : A_{l_i}^{i \in 1 \dots m}\}$. To reiterate, the names of atomic concepts A_x representing variables are unique with respect to the variable and the query. They are not intended to be actually used as syntactic elements in the program. Subsequently, the axioms of K_q are not carried through the program. We ensure that they are taken into account if necessary through rule T-ADD. The rule behaves similarly to the standard subtyping rule T-SUB (see Figure 3.3). It is possible to assign a concept expression D to a term t if it is possible to assign $t : A_x$ using a knowledge base $K \cup \{A_i \sqsubseteq C_i^{i \in 1 \dots n}\}$ and if $K \cup \{A_i \sqsubseteq C_i^{i \in 1 \dots n}\} \models A_x \sqsubseteq D$. To exemplify this, consider a knowledge base consisting of two statements. One saying that the concept expression `Student` is subsumed by `Person` and one saying that the concept expression `Professor` is subsumed by `Person`:

$$K_5 = \{\text{Student} \sqsubseteq \text{Person} \\ \text{Professor} \sqsubseteq \text{Person}\}$$

Furthermore, assume a program that queries for all instances of `Student`, takes the first result and then accesses the projection variable x :

1 `(head (query x ← x type Student)) . x`

This term can be assigned to the type `Student` by extending the knowledge base K with $K_q = \{A_x \sqsubseteq \text{Student}\}$ as is shown in the following derivation tree:

$$\frac{\begin{array}{c} \vdots \\ \emptyset, K_5 \cup (K_q = \{A_x \sqsubseteq \text{Student}\}) \vdash (\text{head (query ...)}) . x : A_x \quad K_5 \cup \{A_x \sqsubseteq \text{Student}\} \models A_x \sqsubseteq \text{Student} \end{array}}{\emptyset, K_5 \vdash (\text{head (query x ← x type Student)}) . x : \text{Student}} \text{T-ADD}$$

The extension of K can be justified by completing the left side of the derivation tree above. Successively applying rules T-PROJ and T-HEAD leads to the rule T-QUERY in which the knowledge base K is actually extended with the new axioms:

$$\frac{\frac{\frac{q = x \leftarrow x \text{ type Student} : K_q = \{A_x \sqsubseteq \text{Student}\} \quad \text{head}(q) = \{x\} \quad K_5 \cup K_q \not\models A_x \sqsubseteq \perp}{\emptyset, K_5 \cup \{A_x \sqsubseteq \text{Student}\} \vdash (\text{query } x \leftarrow x \text{ type Student}) : \text{List } \{x : A_x\}} \text{T-QUERY}}{\emptyset, K_5 \cup \{A_x \sqsubseteq \text{Student}\} \vdash \text{head (query } x \leftarrow x \text{ type Student) : } \{x : A_x\}} \text{T-HEAD}}{\emptyset, K_5 \cup \{A_x \sqsubseteq \text{Student}\} \vdash (\text{head (query } x \leftarrow x \text{ type Student)}) . x : A_x} \text{T-PROJ}$$

As intended, it is impossible to find a derivation tree such that the term is assigned to the concept expression `Professor`.

The subtyping relation mostly relies on the standard rules. As there is no interaction between the newly introduced concept expressions and existing types, it suffices to only define subtyping between concepts. We rely on standard concept inclusion for this. A concept C is a subtype of a concept D

if the knowledge base can show that C is subsumed by D in all possible models (rule S -CONCEPT) according to the knowledge base K . Forwarding this decision to the knowledge base is important as it ensures that implicit knowledge is considered in the decision. Furthermore, we are only concerned with certain knowledge. We treat the case in which it is unknown whether C is included by D as if it is not. To highlight subtyping for concept expressions, let us revisit knowledge base K_5 and the term $(\text{head}(\text{query } x \leftarrow x \text{ type Student})).x$. Given that the term can be assigned to `Student`, we can also find a derivation tree that shows that the term can be assigned to `Person` using rule S -CONCEPT²:

$$\frac{\begin{array}{c} \vdots \\ \emptyset, K_5 \vdash (\text{head}(\text{query } x \leftarrow x \text{ type Student})).x : \text{Student} \end{array} \quad \frac{K_5 \models \text{Student} \sqsubseteq \text{Person}}{K_5 \vdash \text{Student} <: \text{Person}} \text{S-CONCEPT}}{\emptyset, K_5 \vdash (\text{head}(\text{query } x \leftarrow x \text{ type Student})).x : \text{Person}} \text{T-SUB}$$

Algorithmic type checking The type system as described here is directly suited for algorithmic type checking. That is, rules are completely syntax driven with the exception of rule T -ADD (see Algorithm 8). The `TYPEOF` function relies on a knowledge base implementation K that is globally available. In case of a query term, we implement rule T -ADD by actually adding the inferred axioms to K . The `SUBTYPE` procedure receives the knowledge base K as a third parameter. Deciding subtyping between concept types is then simply forwarding the decision to K . Transitivity such as required by rule S -TRANS (see Figure 3.3) is covered by K . The functions for the least upper bound `LUB` and greatest lower bound `GLB` rely on disjunction and conjunction to construct the appropriate concept expressions. There is no interplay between existing types and the newly introduced types (see Algorithm 9).

Currently, there are two different implementations based on the type system described in this Chapter. For one, there is a Java-based implementation [49] which relies on an extended Java Compiler to implement the type checking. For two, there is a Scala-based implementation [85] which uses compiler extensions. Both implementations use the established OWL reasoner [72] for representing the knowledge base K .

4.4. Typecase

Typecase constructs which allow for controlling program flow based on a type of a term are important constructs in programming languages [44]. Examples include standard predicates for testing the runtime type of a value as well as the `try`-statement in Java. The `try`-statement allows proceeding on a case by case basis depending on the type of the exception. It also automatically casts the exception in the different cases. From a type-system perspective, a typecase construct is a type-safe form of

²The term could also be directly assigned to type `Person` via rule T -ADD—we chose rule S -CONCEPT for the sake of the example.

Algorithm 8 Definition of `TYPEOF` and `SUBTYPE` for λ_{DL} .

global variables

K , a knowledge base

function `TYPEOF`(t, Γ)

match t with

case ... then ... \triangleright existing cases

case $t_1 t_2$ when `TYPEOF`(t_1, Γ, K) is $T_1 \rightarrow T'_1$ and `TYPEOF`(t_2, Γ) is T_2
and `SUBTYPE`(T_2, T_1, K) is true then T'_1

case $t_1 = t_2$ when `TYPEOF`(t_1, Γ) is Π_1 and `TYPEOF`(t_2, Γ) is Π_1 then Bool

case $t_1 = t_2$ when `TYPEOF`(t_1, Γ) is C and `TYPEOF`(t_2, Γ) is D then Bool

case $t_1.r$ when `TYPEOF`(t_1, Γ) is C then List $\exists r^- . C$

case query q when `ALLSATISFIABLE`(`AXIOMS`(q), `VARS`(q)) then

$K \leftarrow K \cup \text{AXIOMS}(q)$

$\{v : A_v^{\text{v} \in \text{HEAD}(q)}\}$

case _ then FAIL

function `SUBTYPE`(T, T', K)

match (T, T') with

case ... then ... \triangleright existing cases

case (C, D) when $K \models C \sqsubseteq D$ then true

case _ then false

Algorithm 9 Definition of `LUB` and `GLB` for λ_{DL} .

function `LUB`(T, T')

match (T, T') with

case ... then ... \triangleright existing cases

case (C, D) then $C \sqcup D$

case _ then FAIL

function `GLB`(T, T')

match (T, T') with

case ... then ... \triangleright existing cases

case (C, D) then $C \sqcap D$

case _ then FAIL

casting. We use a typecase for casting graph nodes based on concept expressions. As an example, consider the knowledge base K_4 . The knowledge base states that a `Student` is a `Person` for which a `–studiesAt→` relation pointing to a `University` exists. Likewise, a `Professor` is a `Person` for which a `–worksAt→` relation exists that points to a `University`.

$$K_6 = \{ \text{Student} \sqsubseteq \text{Person} \sqcap \exists \text{studiesAt. University} \\ \text{Professor} \sqsubseteq \text{Person} \sqcap \exists \text{worksAt. University} \}$$

A typecase construct then allows for casting a given `Person` either `Student` or `Professor` and subsequently accessing either the `–studiesAt→` or the `–worksAt→` relation:

```
1  case (head (query x← x type Person)) of
2    type Student as y→ y.studiesAt
3    type Professor as z→ z.worksAt
4    default nil[University]
```

Importantly, knowledge in a knowledge base is not assumed to be complete. That is, even though only `Student` and `Professor` are defined, graph nodes which are instance of `Person` but neither of `Student` nor `Professor` must also be handled. Even more, it may be even unknown whether a graph node belongs to `Professor` or `¬Professor`. Assume the knowledge base K_6 plus the additional statement that there is an object b_1 which is an instance of `Person`. A typecase expression which then covers the cases `Professor` and `¬Professor` looks as follows:

```
1  case b1 of
2    type Professor as x→ x.worksAt
3    type ¬Professor as y→ nil[University]
4    default nil[University]
```

Evaluation proceeds case by case. There is at least one model in which b_1 is not a `Professor`. Therefore, the case in line 2 does not match. The case in line 3 however also does not match—there is at least one model in which b_1 is a `Professor`. Therefore, evaluation resorts to the **default** case.

The typecase construct used in this thesis provides branch control only based on concept expressions (see Figure 4.4). It contains an arbitrary number of cases plus a default case. Branches are evaluated sequentially. If a branch matches, the object is considered typecast into the matched type. It therefore acts as a type-safe casting construct. Evaluation proceeds as follows: The term t_0 is first reduced to an object (rule E-TY). The semantics then tests the object, case by case, until one of them matches (rules E-TY-SUCC and E-TY-FAIL). For each case, the knowledge system decides whether the object is an instance of the concept expression $K \models o_1 : C_1$. If a branch matches, evaluation continues with said branch and other branches are ignored. In case no branch matches, the typecase is reduced to the default case (rule E-TY-DEF).

λ_{DL} with type case

 Extends λ_{DL} (Figure 4.3)

New syntactic elements

$t ::=$... *terms:*
 | **case** t **of** typecase
 \overline{case}
 default t

 $case ::=$ **type** C **as** $x \rightarrow t$ typecase

New evaluation rules

$$\boxed{t \xrightarrow{K} t'}$$

$$\frac{}{\text{case } o_1 \text{ of } \xrightarrow{K} t_1 \quad \text{default } t_1} \quad (\text{E-TY-DEF})$$

$$\frac{K \models o_1 : C_1}{\text{case } o_1 \text{ of } \text{type } C_1 \text{ as } x \rightarrow t_1 \xrightarrow{K} [x \mapsto o_1]t_1 \quad \dots} \quad (\text{E-TY-SUCC})$$

$$\frac{K \not\models o_1 : C_1}{\text{case } o_1 \text{ of } \text{type } C_1 \text{ as } x \rightarrow t_1 \xrightarrow{K} \text{type } C_2 \text{ as } x \rightarrow t_2 \quad \dots} \quad (\text{E-TY-FAIL})$$

New typing rules

$$\boxed{\Gamma, K \vdash t : T}$$

$$\frac{t_1 \xrightarrow{K} t'_1}{\text{case } t_1 \text{ of } \text{type } C_1 \text{ as } x \rightarrow t_2 \xrightarrow{K} \text{case } t'_1 \text{ of } \text{type } C_1 \text{ as } x \rightarrow t_2 \quad \dots} \quad (\text{E-TY})$$

$$\frac{\begin{array}{l} \Gamma, K \vdash t_0 : C_0 \\ (\Gamma, x_i : C_i), K \vdash t_i^{i \in 1 \dots n} : T \\ \Gamma, K \vdash t_{n+1} : T \\ K \not\models C_0 \sqcap C_i \sqsubseteq \perp \\ K \not\models C_i \sqsubseteq C_j \text{ for } i < j \end{array}}{\text{case } t_0 \text{ of } \text{type } C_1 \text{ as } x_1 \rightarrow t_1 \quad \Gamma, K \vdash \text{type } C_2 \text{ as } x_2 \rightarrow t_2 : T \quad \dots \quad \text{default } t_{n+1}} \quad (\text{T-TYPECASE})$$

Figure 4.4.: Syntax and Evaluation rules of the typecase expression.

The typing rule for the typecase expression is also shown in Figure 4.4. The term t_0 must be of type C_0 , i. e., a concept expression. The types for the non-default cases are determined in a context where the variable x_i is bound to the type C_i of the case. The idea is that t_0 is cast to C_i type-safely and to be accessed as x_i within t_i . The result type of the typecase is the least upper bound of the types of all cases including the default case. There are additional premises to ensure meaningful cases. That is, the intersection between all C_i and C_0 should be satisfiable, as it would then be impossible for a case to ever match. Also, a case should never subsume a preceding case, as cases are tried sequentially.

4.5. Type Safety

Given the design choices of λ_{DL} , the language is type-safe with the routine exceptions of trying to access **head** or **tail** of empty lists. To reiterate, if a term t has a type T , then this means that t evaluates to a value belonging to T . Likewise for queries. To show type safety, we first show that the axioms inferred a query are sound. That is, the constraints inferred for each variable are satisfied for all query results. Using this, we can then proceed to show progress and preservation for λ_{DL} .

4.5.1. Soundness of Query Typing

Typing queries is sound if the constraints inferred for each variable are satisfied for all possible mappings of the variable. To reiterate, each variable x_i of a query q is represented through one or several axioms of the form $A_{x_i} \sqsubseteq C$. All potential query results for the variable x_i must be instances of the concept expression C . If this is the case, then the atomic concept A_{x_i} represents a set of graph nodes that contains all potential query results for the variable x_i and is therefore an approximation of the evaluation results for the variable.

Definition 8 (Soundness of axiom inference). *Given a knowledge base K , a conjunctive query q with its variables $\text{Vars}(q)$ and the inferred set of axioms K_q which contains axioms of the form $A_{x_i} \sqsubseteq C_{x_i}$ (see Figure 4.2). The inferred axioms are sound if:*

$$\forall x \in \text{Vars}(q) \in \forall \mu : \llbracket q \rrbracket_K : K \cup K_q \models \mu(x) : C_{x_i}^{i \in 1 \dots n}$$

We now show that our typing relation “:” as defined in Figure 4.2 is sound.

Theorem 3. *For any knowledge base K and conjunctive query q , the typing relation assigns concept expressions to variables such that axiom inference is sound.*

Proof. We show this by induction on the evaluation rules of $\llbracket q = \bar{x} \leftarrow gp \rrbracket_K$.

Q-SVAR $gp = x_1 r o'$, $gp : \{a_{x_1} \sqsubseteq \exists r.\{o'\}\}$.

Evaluation of gp returns all o for which $(o, o') \in r(K)$. Any model must interpret the concept expression $\exists r.\{o'\}$ as $(\exists r.\{o'\})^I = \{o \mid (o, o'') \in r^I \wedge o'' \in \{o'^I\}\}$ (see Figure 2.7). Therefore, $o \in (\exists r.\{o'\})^I$ holds for all models.

Q-OVAR Similar to case Q-SVAR.

Q-VARS $gp = x_1 r x_2$, $gp : \{A_{x_1} \sqsubseteq \exists r.A_{x_2}, A_{x_2} \sqsubseteq \exists r^-.A_{x_1}\}$.

Evaluation of gp returns all $(o, o') \in r(K)$. Any such node o must be in the interpretation of $\exists r.A_{x_2}$ because all o' must be in the interpretation of A_{x_2} . Likewise, for o' .

Q-CONJ $gp = gp_1 \wedge gp_2$, $gp_1 : K_1^q, gp_2 : K_2^q$, $gp : K_1^q \cup K_2^q$.

By induction hypothesis, inferred axioms are sound for gp_1 and K_1^q as well as gp_2 and K_2^q . DL is monotonous—taking the union $K_{q_1} \cup K_{q_2}$ preserves all inferences.

Q-PROJ Immediate since the inferred concepts are not modified.

□

As a consequence, axiom inference is sound. That is, given a knowledge base K and a query q as well as the inferred set of axioms $K_q = \{A_{x_i} \sqsubseteq C_{x_i}^{x_i \in \text{Vars}(q)}\}$, it holds that $\forall x_i \in \text{Vars}(q) : \forall \mu \in \llbracket q \rrbracket_K : K \cup K_q \models \mu(x_i) : C_{x_i}$.

4.5.2. Soundness of the Type System

Given the design choices of λ_{DL} , the language is type-safe. A well-typed program does not get stuck. As discussed in Section 3.2.3, the only exception to this concerns lists. We therefore show that if a program is well-typed, then the only way it can get stuck is by reaching a point where it tries to compute `head nil[T]` or `tail nil[T]`. As introduced in Section 2.1.3, we proceed in two steps by showing that a well-typed term is either a value or it can take a step (progress) and by showing that if that term takes a step, the result is also well-typed (preservation). We start by observing that a well-typed value of type C must always be a graph node:

Lemma 2 (Canonical forms of λ_{DL}). *Let v be a well-typed value. Then one of the following must be true:*

1. *If v is a value of type `Bool`, then either $v = \text{true}$ or $v = \text{false}$.*
2. *If v is a value of type `Nat`, then v is a numerical value nv according to the grammar defined in Figure 3.1.*
3. *If v is a value of type `C`, then v is a graph node `o`.*

4. If v is a value of type $\{l_i : T_i^{i \in 1 \dots n}\}$ then v must be a record of the form $\{l_j = t_j^{j \in 1 \dots m}\}$ with $\{l_i\} \subseteq \{l_j\}$ and $\Gamma, K \vdash t_i : T_i$ for all cases in which $i = j$.
5. If v is a value of type $\text{List } T$, then v is either an empty list $\text{nil}[T]$ or of the form $\text{cons } v_1 \dots \text{nil}[T_1]$ with $T_1 <: T$ and $\Gamma, K \vdash v_1 : T$.
6. If v is a value of type $T \rightarrow T'$, then v is a λ -abstraction $\lambda x : T_1 . t_2$ with $T_1 <: T$ and $\Gamma, x : T_1, K \vdash t_2 : T'$.

Given Lemma 2, we can show that a well-typed term is either a value or it can take a step. Given the design choices of λ_{DL} , this is straightforward. In particular, there may be *unknown facts* which are true in some models and false in others. λ_{DL} treats those as false. We also foresee that no case of typecase fits to the run-time value and thus insist on the default case.

Theorem 4 (Progress in λ_{DL}). *Let t be a well-typed closed term. If t is not a value, then there exists a term t' such that $t \xrightarrow{K} t'$. If $\Gamma, K \vdash t : T$, then t is either a value, a term containing the forms $\text{head nil}[T]$ and $\text{tail nil}[T]$, or there is some t' with $t \xrightarrow{K} t'$.*

Proof. By induction on the derivation of $\Gamma, K \vdash t : T$. We omit the cases of the proof that are unchanged compared to the standard λ -calculus and instead focus on the cases that are specific to our language.

T-RPROJ $t = t_1.r$, $\Gamma, K \vdash t_1 : C$, $\Gamma, K \vdash t : \text{List } \exists r^-.C$.

By hypothesis, t_1 is either a value or it can take a step. If it can take a step, E-RPROJ applies. If it is a value, then by Lemma 2, $t_1 = o_1$, therefore rule E-PROJROLE applies.

T-EQ-NOM $t_1 = t_2$, $\Gamma, K \vdash t_1 : C_1$, $\Gamma, K \vdash t_2 : C_2$.

By hypothesis, both t_1 and t_2 are either values or they can take a step. If one of the two can take a step, then either rule E-EQ1 or rule E-EQ2 applies. If both are values, then by Lemma 2, they are both graph nodes ($t_1 = o_1$ and $t_2 = o_2$). Therefore, either rule E-EQ-NODE or E-NEQ-NODE applies.

T-EQ-PRIM $t_1 = t_2$, $\Gamma, K \vdash t_1 : \Pi_1$, $\Gamma, K \vdash t_2 : \Pi_2$.

By hypothesis, both t_1 and t_2 are either values or they can take a step. If one of the two can take a step, then either rule E-EQ1 or rule E-EQ2 applies. If both are values, then they must be primitive values ($t_1 = \pi_1$ and $t_2 = \pi_2$). Therefore, either rule E-EQ-PRIM or E-NEQ-PRIM applies.

T-NOMINAL $t = o_1$, $\Gamma, K \vdash o_1 : \{o_1\}$.

Immediate since $t = o_1$ is a value.

T-QUERY $t = \text{query } q$, $\Gamma, K \vdash t : \text{List } \{l_i : C_i^{i \in 1 \dots n}\}$.

Immediate since rule E-QUERY applies.

T-ADD Results follow from the induction hypothesis since rule T-ADD requires a term to be well-typed.

T-TYPECASE $t = \text{case } t_0 \text{ of}$

$\overline{\text{cases}}$

default $t_{n+1}, \quad \Gamma, K \vdash t_0 : C_0, \quad \Gamma, K \vdash t : T.$

By hypothesis, t_0 is either a value or it can take a step. If it can take a step, then rule E-TY applies. If it is a value, then by Lemma 2, $t_0 = o_0$. If $\overline{\text{cases}}$ is non-empty, then either rule E-TY-SUCC or rule E-TY-FAIL applies. If it is empty, then rule E-TY-DEF applies.

□

Substitution in λ_{DL} does not differ from standard approaches, e.g., as described in [81]. Substitution on terms does preserve the type. We continue to show that if a term takes a step by the evaluation rules, its type is preserved.

Theorem 5 (Preservation in λ_{DL}). *Let t be a term and T a type. If a type is assigned to t , written $\Gamma, K \vdash t : T$ and $t \xrightarrow{K} t'$, then $\Gamma, K \vdash t' : T$.*

Proof. By induction on the derivation of $\Gamma, K \vdash t : T$. Again, we only examine the specific cases specific to our language. We omit cases that are unchanged compared to the standard λ -calculus.

T-RPROJ $t = t_1.r, \quad \Gamma, K \vdash t_1 : C_1, \quad \Gamma, K \vdash t : \text{List}(\exists r^-.C_1)$

There are 2 different cases by which t' can be derived:

E-PROJROLE $t' = \text{cons } \mu_1(x_1) \dots$ with $K \models o_1 : C_1$ and $\llbracket x_1 \leftarrow o_1 r x_1 \rrbracket_K = \mu_i^{i \in 1 \dots n}$.

The concept expression $\exists r^-.C_1$ is the set of graph nodes with an incoming relation via r from a node belonging to C_1 . Since $K \models o_1 : C_1$ all o_2 reachable via r from o_1 must be an instance of this concept. Therefore, $t' : \text{List}(\exists r^-.C_1)$.

E-RPROJ $t' = t'_1.r.$

By hypothesis, $t \xrightarrow{K} t'$ preserves the type. Rule T-RPROJ therefore still applies.

T-EQ-NOM $t = t_1 = t_2. \quad \Gamma, K \vdash t_1 : C_1, \quad \Gamma, K \vdash t_2 : C_2, \quad \Gamma, K \vdash t : \text{Bool}.$

There are 4 different rules by which t' can be derived:

E-EQ1 $t' = t'_1 = t_2.$

By hypothesis, $t_1 \xrightarrow{K} t'_1$ preserves the type. Therefore, via rule T-EQ-NOM, $t' : \text{Bool}$.

E-EQ2 $t' = v_1 = t'_2.$

By hypothesis, $t_2 \xrightarrow{K} t'_2$ preserves the type. By rule T-EQ-NOM, $\Gamma, K \vdash t' : \text{Bool}$.

E-EQ-NODE $t' = \text{true}.$

By rule T-TRUE, $\Gamma, K \vdash t' : \text{Bool}$.

E-NEQ-NODE $t' = \text{false}$.

By rule T-FALSE, $\Gamma, K \vdash t' : \text{Bool}$.

T-EQ-PRIM $t = t_1 = t_2$. $\Gamma, K \vdash t_1 : \Pi_1$, $\Gamma, K \vdash t_2 : \Pi_2$, $\Gamma, K \vdash t : \text{Bool}$.

There are 4 different rules by which t' can be derived:

E-EQ1 $t' = t'_1 = t_2$.

By hypothesis, $t_1 \xrightarrow{K} t'_1$ preserves the type. Therefore, via rule T-EQ-PRIM, $\Gamma, K \vdash t' : \text{Bool}$.

E-EQ2 $t' = v_1 = t'_2$.

By hypothesis, $t_2 \xrightarrow{K} t'_2$ preserves the type. By rule T-EQ-PRIM, $\Gamma, K \vdash t' : \text{Bool}$.

E-EQ-PRIM $t' = \text{true}$.

By rule T-TRUE, $\Gamma, K \vdash t' : \text{Bool}$.

E-NEQ-PRIM $t' = \text{false}$.

By rule T-FALSE, $\Gamma, K \vdash t' : \text{Bool}$.

T-NOMINAL $t = o$, $\Gamma, K \vdash o : \{o\}$.

Vacuously fulfilled since o is a value—therefore, it cannot be that there is a t' such that $t \xrightarrow{K} t'$.

T-QUERY $t = \text{query } q$, $q : K_q$, $\Gamma, K \cup K_q \vdash t : \text{List } \{l_i : A_i^{1 \in 1 \dots m}\}$ with $\{\check{a}l_i^{1 \in 1 \dots m}\}$ being the head of q .

If t takes a step, then it can only be through rule E-QUERY. By Theorem 3, all graph nodes in the evaluation results are instances of their respective concepts. Furthermore, by rules T-RCD and T-LIST the result is a list of records. The type is therefore preserved.

T-ADD Results follow from the induction hypothesis since rule T-ADD requires a term to be well-typed.

T-TYPECASE $t = \text{case } t_0 \text{ of}$

type C_1 **as** $x_1 \rightarrow t_1$

type C_2 **as** $x_2 \rightarrow t_2$

...

default t_{n+1}

$\Gamma, K \vdash t_0 : C$, $\Gamma, K \vdash t_i : T^{i \in 1 \dots n+1}$.

There are four different rules by which t' can be derived:

E-TY-DEF $t' = t_{n+1}$. Through rule T-TYPECASE, $t_{n+1} : T$, therefore $\Gamma, K \vdash t' : T$.

E-TY-SUCC $t' = t_i$ with $i \in \{1 \dots n\}$. Again, rule T-TYPECASE enforces $t_i : T$, therefore $\Gamma, K \vdash t' : T$.

E-TY-FAIL The rule removes one case that did not match. The remaining cases are still of type T .

Rule T-TYPECASE is still applicable and $\Gamma, K \vdash t' : T$ still holds.

E-TY Term t_0 takes a step and becomes t'_0 . By induction hypothesis, this preserves the type. Rule **T-TYPECASE** is still applicable, $\Gamma, K \vdash t' : T$ therefore holds.

□

As a direct consequence of Theorems 4 and 5, a well-typed, closed term does not get stuck during evaluation. The only exception concerns handling of lists which can get stuck if `head` or `tail` is applied to an empty list. To a certain degree, type safety holds even when the knowledge system is evolving. Additional statements are unproblematic. Description logics are monotonous—additions do not invalidate existing inferences. Deletion and modification of the actual data (A-Box) is unproblematic unless the program contains statements explicitly referencing the objects under modification. Of course, type safety cannot be guaranteed if schematic parts (T-Box) of the knowledge system are altered.

4.6. Summary and Discussion

In this Chapter, we have studied λ_{DL} —a typed λ -calculus for the Semantic Web that is built around concept expressions as types as well as queries. We have shown that by using conceptualizations as they are defined in the knowledge system itself, type safety can be achieved. This helps in writing less error-prone programs, even when facing knowledge systems. However, it is also noteworthy that the open-world semantics used by description logics can be problematic. While it allows for modeling of unknown facts, it may also indicate the presence of relations that may not actually be accessible in the data. A fixed-domain semantics as employed by SHACL that is closer to the behavior of existing type systems will be explored in Chapter 5.

Type Checking with SHACL

While type checking with description logics does provide a certain level of safety, its open-world semantics is problematic. While it may be conceptually guaranteed that all instances of `(Student)` have a `-hasName→` relation, there may be nodes such as `(bob)` for which the name is not known. SHACL on the other hand provides a fixed-domain semantics using integrity constraints. A SHACL shape can guarantee that relationships are known for all graph nodes that conform to the shape. We therefore introduce λ_{SHACL} , a research language that features type-safe programming by using SHACL shapes during the type checking process. This chapter is based on [68].

5.1. Design Principles and Example Use Case

Key Design Principles The language λ_{SHACL} is based on several key design principles:

Type checking is defined with respect to a set of SHACL shapes. Type checking relies on a set of SHACL shapes that provide schematic information about RDF graphs. Type checking then uses this set of shapes to show that the program will not abort with a run-time error for all graphs that conform to the set of shapes.

Programs are defined with respect to a RDF data graph. Evaluation of a program is defined with respect to an RDF data graph. RDF data graphs that are used during evaluation are expected to conform to the SHACL shapes used during type checking.

Shape names as types. Again, types represent sets of values (see Section 2.1.2). We rely on shapes that represent sets of graph nodes. A shape is a triple (s, ϕ, \hat{q}) consisting of a shape name s , a constraint ϕ and a query for target nodes \hat{q} . We use the shape name s as a syntactic symbol for a type which represent the set of graph nodes that conform to that shape.

Shape Containment as subtyping. Integration of shapes as types calls for a subtype relation between this new form of types. As the subtype relation can be seen as a subset relation between sets of values, we use shape containment. That is, showing that all nodes conforming to one shape must also conform to the other shape for all possible data graphs.

Typing of queries. Again, queries constitute the main form of data access. As such, they must be assigned meaningful types representing the graph nodes that the query evaluates to. In particular, we infer a set of shapes from the query.

Example Use Case As an example, consider an application that is defined with respect to the set of shapes as shown in Figure 5.1: `StudentShape` targets all instances of `Student` (line 2). The shape enforces that students are also instance of `Person` (line 3). It also enforces the presence of at least one `-studiesAt→` relation pointing to a node conforming to `UniversityShape` (lines 4–8). `UniversityShape` has no target nodes, but enforces nodes to be instances of `University` (line 11) as well as the presence of at least one `-hasLocation→` relation (lines 12–15). Lastly, `PersonShape` targets all instances of `Person` (line 17). A person must have exactly one `-hasName→` relation (lines 18–23). In a slight simplification, we constrain all nodes reachable through the `-hasName→` relation such that they conform to a shape named `StringShape` (line 22) which represents string values¹. The example application is defined with respect to RDF data

```

1  ex:StudentShape a sh:NodeShape;
2  sh:targetClass ex:Student;
3  sh:class ex:Person;
4  sh:property [
5    sh:path ex:studiesAt;
6    sh:minCount 1;
7    sh:node ex:UniversityShape
8  ].
9
10 ex:UniversityShape a sh:NodeShape;
11 sh:class ex:University.
12 sh:property [
13   sh:path ex:hasLocation;
14   sh:minCount 1
15 ];
16 ex:PersonShape a sh:NodeShape;
17 sh:targetClass ex:Person;
18 sh:property [
19   sh:path ex:hasName;
20   sh:minCount 1;
21   sh:maxCount 1;
22   sh:node ex:StringShape;
23 ].
24
25
26
27
28
29
30
```

Figure 5.1.: Example SHACL shape graph.

graphs that conform to the shape graph. One such data graph is the empty graph. Other examples of graphs that conform to the SHACL shape graphs are graphs G_4 , G_5 and G_6 as shown in Figure 5.2. All of those graphs are potential inputs for the program.

¹SHACL allows for constraints on literal values through XSD datatypes and the `sh:datatype` constraint. As XSD datatypes can typically be mapped to types in standard programming languages, we abstracted them away for simplicity.

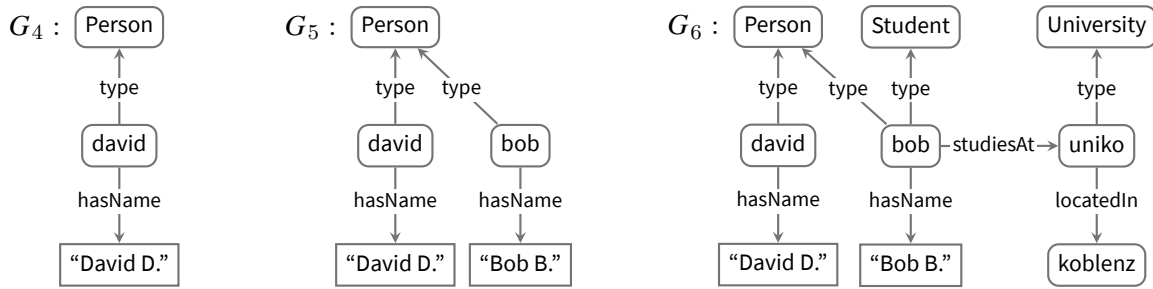


Figure 5.2.: Examples for graphs that conform to the shape graph in Figure 5.2.

Our example application should implement three different functions: First, it should query for all instances of `Student`. Second, it should define a function that is able to return the universities where the student studies. Third, it should define a function that returns the name of a given person.

Querying for all instances of `Student` is again straightforward as queries are integrated into the language:

```
1 query x ← x type Student
```

The expression will evaluate to a list of records where each record has a single label representing the projection variable `x`. The type of the record label is equivalent to `StudentShape`. A function that returns the list of universities where a student studies can be defined as follows:

```
1 let getUniversities = λ (student:StudentShape) . student.studiesAt
```

The domain of the `getUniversities` function is the shape `StudentShape`. In the body of the function, a role projection can be used to traverse the graph from the given node to all nodes reachable via the `-studiesAt→` relation. The SHACL shape graph shown in Figure 5.1 guarantees the presence of at least one `-studiesAt→` relation. However, it does not specify a maximum number of nodes that can be reached through the relation. For all nodes that can be reached through the relation, it is known that they conform to `UniversityShape`. Subsequently, the function is of type `getUniversities : StudentShape → List UniversityShape`. Lastly, a function that returns the name of a given person can be defined as follows:

```
1 let getName = λ (person:PersonShape) . person.hasName
```

The input type of the function is the shape `PersonShape`. Again, we use role projection to access the name. However, for `PersonShape`, the SHACL shape graph guarantees that there is exactly one successor via the `-hasName→` relation. Subsequently, the type of the function is `getName : PersonShape → StringShape`. Importantly, nodes conforming to `StudentShape` must also conform to `PersonShape`. `PersonShape` targets all instances of `Person` whereas the constraints of `StudentShape` enforce students to be instances of `Person`. Subsequently, the function can be applied to results of the query defined above.

5.2. Types for Conjunctive Queries

To provide a typed integration for conjunctive queries, we must again reconstruct types based on the query. As shapes constitute types in λ_{SHACL} , we reconstruct a set of shapes from the query. We represent each variable through a shape name. For this, we assume the possibility to transform each variable x of a query into a globally unique shape name s_x . Variables which share a name but occur in different queries should therefore be represented by different concepts. We then examine each graph pattern (abbreviated as gp) of the query $\bar{x} \leftarrow gp$ and reconstruct constraints ϕ for the shape names based on the graph pattern. The result of this process is a set of shapes S_q .

Typing rules for SPARQL CQs		$q : S_q$	
$x p o : \{(s_x, \geq_1 p.o, \perp)\}$	(QT-ROLE1)		$x_1 p x_2 : \{(s_{x_1}, \geq_1 p.s_{x_2}, \perp),$ $(s_{x_2}, \geq_1 p^-.s_{x_1}, \perp)\}$
$o p x : \{(s_x, \geq_1 p^-.o, \perp)\}$	(QT-ROLE2)		
			$\frac{gp_1 : S_1 \quad gp_2 : S_2}{gp_1 \wedge gp_2 : S_1 \hat{\wedge} S_2}$ (QT-CONJ)
			$\frac{gp : S}{\bar{x} \leftarrow gp : S}$ (QT-PROJ)

Figure 5.3.: Rules for assigning shapes to variables in queries.

Our typing relation $q : S_q$ (see Figure 5.3 for the conjunctive query q constructs the set S_q in the following manner: For every subject var pattern $x p o$ in the body of q we assign the constraint $\geq_1 p.o$ (rule QT-ROLE1). Likewise for object var patterns $o p x$, although we use the constraint $\geq_1 p^-.o$ (QT-ROLE2). In case of variables used in both, subject and object positions $x_1 p x_2$, we infer two shapes s_{x_1} and s_{x_2} . We use shape references to express the dependencies and infer the constraints $\geq_1 p.s_{x_2}$ and $\geq_1 p^-.s_{x_1}$ (rule QT-ROLE3). We do not use target nodes when constructing the shapes. As described in Section 2.2.4, constraints define the set of nodes that conform to the shape whereas target nodes only impact the validity of the graph. We therefore always use \perp to denote that the shapes have no target nodes. In case of a conjunction of graph patterns (denoted by $gp_1 \wedge gp_2$), we infer the sets of shapes for each query body individually and then combine the results using the operator $\hat{\wedge}$ (rule QT-CONJ). The relation $\hat{\wedge}$ takes two sets of shapes S_1 and S_2 and combines them into a unique set performing a full outer join on the shape names. That is, if a shape name occurs in both sets, then constraint conjunction is used to combine them. Otherwise, the inferred constraint is simply carried

along:

$$\begin{aligned} S_1 \hat{\wedge} S_2 = & \{(s_x, \phi_1 \wedge \phi_2, \perp) \mid (s_x, \phi_1, \perp) \in S_1 \wedge (s_x, \phi_2, \perp) \in S_2\} \cup \\ & \{(s_x, \phi_1, \perp) \mid (s_x, \phi_1, \perp) \in S_1 \wedge (s_x, \phi_2, \perp) \notin S_2\} \cup \\ & \{(s_x, \phi_2, \perp) \mid (s_x, \phi_1, \perp) \notin S_1 \wedge (s_x, \phi_2, \perp) \in S_2\} \end{aligned}$$

As an example, consider the query

$$q_1 = x \leftarrow x \text{ type Student} \wedge x \text{ studiesAt } y$$

The query contains the two graph patterns $gp_1 = x \text{ type Student}$ and $gp_2 = x \text{ studiesAt } y$. Both patterns are examined individually (rule QT-CONJ). In case of gp_1 , rule QT-ROLE1 applies. That is, the pattern $x \text{ type Student}$ is assigned to the set of shapes $\{(XShape, \geq_1 \text{ type.Student}, \perp)\}$. Second, the graph pattern $x \text{ studiesAt } y$ is assigned to the set of shapes $\{(XShape, \geq_1 \text{ studiesAt.YShape}, \perp), (YShape, \geq_1 \text{ studiesAt}^-.XShape, \perp)\}$. Rule QT-CONJ then combines the two sets of shapes using $\hat{\wedge}$, yielding the following set of shapes:

$$q_1 : S_{q_1} = \left\{ \begin{array}{l} (XShape, \geq_1 \text{ type.Student} \wedge \geq_1 \text{ studiesAt.YShape}, \perp), \\ (YShape, \geq_1 \text{ studiesAt}^-.XShape, \perp) \end{array} \right\}$$

5.3. Core Language

Syntax and Semantics The language λ_{SHACL} (see Figure 5.4) is an extension of the standard λ -calculus as presented in Chapter 3. New terms (denoted by t) include the **query** keyword for querying an RDF graph as well as a projection from a graph node onto a list of graph nodes via a property p . Values (denoted by v) include graph nodes o whereas shape names s are used as a new form of types (denoted by T).

The operational semantics bear no significant difference to the standard ones as used in Chapter 3. We define evaluation rules $t \xrightarrow{G} t'$ with respect to an RDF graph G which has no impact for all constructs unrelated to RDF graphs. We therefore omit them and focus on the newly added constructs for querying and role projection. A role projection term $t_1.p$ first reduces the term t_1 to a value (rule E-RPROJ). If this value is a graph node o_1 the term $o_1.p$ can be evaluated via a query using the graph pattern $o_1 p x_1$ and the results can be returned as a list (rule E-PROJROLE). Querying data via the **query**-keyword evaluates the query q over the RDF data graph G . This results in a set of mappings μ . These mappings are then converted into a list of records by taking each projection variable and turning it into a label of a record. The value referenced by the record label is then the graph node to which a given μ maps the variable (rule E-QUERY). As with λ_{DL} (c. f. Section 4.3), we chose to convert query results into a list of records rather than a set even though this creates an implicit ordering. Lists are

a more basic programming language construct and subsequent processing of query results introduces an ordering anyways.

λ_{SHACL}	Extends λ_{Full} (Fig. 3.6)
<p><i>New syntactic elements</i></p> <p>$t ::= \dots$ <i>terms:</i></p> <p style="padding-left: 20px;"> query q SPARQL query</p> <p style="padding-left: 20px;"> $t.p$ role projection</p> <p>$v ::= \dots$ <i>values:</i></p> <p style="padding-left: 20px;"> o graph node</p> <p>$T ::= \dots$ <i>types:</i></p> <p style="padding-left: 20px;"> s shape name</p>	<p><i>New typing rules</i> $\Gamma, S \vdash t : T$</p> $\frac{\Gamma, S \vdash t_1 : s_1 \quad \text{genName}() = s'_1 \quad S \cup \{(s_{\text{tmp}}, \geq_1 p.\top, \perp)\} \vdash s_1 <: s'_1}{\Gamma, S \cup \{s'_1, \geq_1 p^-.s_1, \perp\} \vdash t_1.p : \text{List } s'_1} \quad (\text{T-RPROJ})$ $\frac{q : S_q \quad \text{Head}(q) = \{l_i^{i \in 1 \dots m}\}}{\Gamma, S \cup S_q \vdash \text{query } q : \text{List } \{l_i : s_i^{i \in 1 \dots m}\}} \quad (\text{T-QUERY})$ $\frac{\Gamma, S \cup \{(s_i, \phi_i, \perp)^{i \in 1 \dots n}\} \vdash t : s \quad S \cup \{(s_i, \phi_i, \perp)^{i \in 1 \dots n}\} \vdash s <: s'}{\Gamma, S \vdash t : s'} \quad (\text{T-ADD})$
<p><i>New evaluation rules</i> $t \xrightarrow{G} t'$</p> $\frac{\llbracket x_1 \leftarrow o_1 p x_1 \rrbracket_G = \{\mu_i^{i \in 1 \dots n}\}}{o_1.p \xrightarrow{G} \text{cons } \mu_1(x_1) \dots \text{cons } \mu_n(x_1) \text{ nil}} \quad (\text{E-PROJROLE})$ $\frac{t_1 \xrightarrow{G} t'_1}{t_1.p \xrightarrow{G} t'_1.p} \quad (\text{E-RPROJ})$ $\frac{\llbracket q \rrbracket_G = \{\mu_i^{i \in 1 \dots n}\} \quad q = l_j^{j \in 1 \dots m} \leftarrow gp}{\text{query } q \xrightarrow{G} \text{cons } \{l_j = \mu_1(l_j)^{j \in 1 \dots m}\} \dots \text{cons } \{l_j = \mu_n(l_j)^{j \in 1 \dots m}\} \text{ nil}} \quad (\text{E-QUERY})$	<p><i>New subtyping rules</i> $S \vdash T <: T'$</p> $\frac{\forall G \in \mathcal{G} : \forall \sigma \in \text{Faith}(G, S) : \quad \forall o \in \text{Nodes}(G) : s_1 \in \sigma(o) \Rightarrow s_2 \in \sigma(o)}{S \vdash s_1 <: s_2} \quad (\text{S-SHAPE})$

Figure 5.4.: Syntax, Semantics and type system rules for λ_{SHACL} .

Type System and Subtyping The addition of shape names s as types is the most distinguishing feature of the type system. Besides the context Γ , we also require a set of shapes S . As we use only shape names as types, S allows for accessing the full definition of a shape including its constraints.

For normal constructs, the addition of S has little impact. We omit these rules and focus on the ones specific to constructs related to RDF graphs. A noteworthy detail is that we do not include any rule for typing single graph nodes o . Programs that directly mention graph nodes are therefore not allowed by the type system. In case of role projections $t_1.p$, the term t_1 must evaluate to a graph node—that is, it must be typed with a shape name s_1 (rule T-RPROJ). In order to check whether the projection is possible, we test whether s_1 is a subtype of a shape s_{tmp} that has the constraint $\geq_1 p.\top$. We also generate a fresh shape name s'_1 using a function `genShapeName` that acts as a name for the constraint $\geq_1 p^-.s_1$. The result of the operation can then be typed using the type `List s'_1`. In case the term **query** q , q is typed using the rules as described in Section 5.2 (rule T-QUERY). The record type, representing an individual mapping, is then build using the shape names s_{l_i} for each variable l_i . The resulting type is then a list of this record type. Any shape name that is generated by typing projections or typing queries is not intended to be used as syntactic elements of the program. As we did in case of λ_{DL} (c. f. Section 4.3), we rely on rule T-ADD to consider them when needed. To exemplify this, reconsider the set of shapes as defined in Figure 5.1, given in abstract syntax:

$$S_1 = \{ \begin{array}{l} (\text{StudentShape}, \quad \geq_1 \text{type.Person} \wedge \geq_1 \text{studiesAt}.\top \\ \quad \wedge \forall \text{studiesAt.UniversityShape}, \quad x \leftarrow x \text{ type Student}), \\ (\text{PersonShape}, \quad =_1 \text{hasName.StringShape}, \quad x \leftarrow x \text{ type Person}), \\ (\text{UniversityShape}, \quad \geq_1 \text{locatedIn}.\top, \quad \perp) \end{array} \}$$

Furthermore, a query that retrieves all instances of `Student` is assigned the following set of shapes:

$$q_2 = x \leftarrow x \text{ type Student} : S_{q_2} = \{(XShape, \geq_1 \text{type.Student}, \perp)\}$$

Using rule T-ADD, a term that queries for all instances of `Student`, takes the head of the resulting list, and accesses the projection variable x , can be typed with `StudentShape`:

$$(\text{head} (\text{query } x \leftarrow x \text{ type Student})).x : \text{StudentShape}$$

This is witnessed by the following derivation tree:

$$\frac{\begin{array}{c} \vdots \\ \Gamma, S_1 \cup S_{q_2} \vdash (\text{head} (\text{query} \dots)).x : XShape \end{array} \quad \begin{array}{c} \vdots \\ S_1 \cup S_{q_2} \vdash XShape <: \text{StudentShape} \end{array}}{\Gamma, S_1 \vdash (\text{head} (\text{query } x \leftarrow x \text{ type Student})).x : \text{StudentShape}} \text{T-ADD}$$

Intuitively, all nodes conforming to `XShape` must be contained in `StudentShape`. Nodes conforming to `XShape` must be instances of `Student`. Those are the exact target nodes for `StudentShape`. The extension of the set of shapes as used by rule T-ADD in the example is justified as witnessed by the following

derivation tree:

$$\frac{\frac{\frac{x \leftarrow x \text{ type Student} : S_{q_2} \quad \text{Head}(x \leftarrow x \text{ type Student}) = \{x\}}{\Gamma, S_1 \cup S_{q_2} \vdash \mathbf{query} \ x \leftarrow x \text{ type Student} : \text{List} \ \{x : X\text{Shape}\}} \text{T-QUERY}}{\Gamma, S_1 \cup S_{q_2} \vdash \mathbf{head} \ (\mathbf{query} \ x \leftarrow x \text{ type Student}) : \{x : X\text{Shape}\}} \text{T-HEAD}}{\Gamma, S_1 \cup S_{q_2} \vdash (\mathbf{head} \ (\mathbf{query} \ x \leftarrow x \text{ type Student})).x : X\text{Shape}} \text{T-PROJ}}$$

For subtyping, a single additional case (denoted by rule s-SHAPE) suffices as there is no interaction between the newly added shape names and existing types. Subtyping between two shapes s and s' is intricate as the type system must rely on certain knowledge—meaning that for all RDF data graphs that conform to the set of shapes, it must be true that the nodes conforming to s are a subset of the nodes conforming to s' . In other words, s must be contained in s' . Subtyping between shapes therefore relies on the infinitely large set of all RDF data graphs \mathcal{G} and the set of all faithful assignments $\text{Faith}(G, S)$ for the individual graphs and the set of shapes S . For each of those assignments, it must be the case that conforming to shape s implies that the graph nodes also conform to shape s' .

To exemplify this, consider the relationship between the shapes StudentShape and PersonShape in the set of shapes S_1 . Those two shapes must be in a subtype relation:

$$\frac{\forall G \in \mathcal{G} : \forall \sigma \in \text{Faith}(G, S) : \forall o \in \text{Nodes}(G) : \text{StudentShape} \in \sigma(o) \Rightarrow \text{PersonShape} \in \sigma(o)}{S_1 \vdash \text{StudentShape} <: \text{PersonShape}} \text{s-SHAPE}$$

The constraints of StudentShape enforces its nodes to be instances of Person . PersonShape targets all instances of Person . As a consequence, it is impossible for any faithful assignment to assign StudentShape but not PersonShape to a graph node.

Algorithmic Type Checking With the exception of rule s-SHAPE, the type system is directly suited for algorithmic type checking (see Algorithm 10).

As we did for λ_{DL} (c. f. Section 4.3), we use a global state to represent the set of shapes which is modified during the type checking process. Rules T-RPROJ and T-QUERY then simply add new cases to the TYPEOF function, whereas the reconstruction of shapes from queries is implemented in a function SHAPES. The SUBTYPE function however is problematic as shape containment is an open issue. To the best of our knowledge, there are no sound and complete algorithms for deciding shape containment. A trivial approach that is sound but incomplete is to only allow for conjunction and no negation in constraints, essentially allowing for constraints to be seen as sets. That is, we define a subset of SHACL where constraints are built according to the following grammar:

$$\begin{aligned} \phi_{\text{set}} &::= \phi_{\text{basic}} \mid \phi_{\text{set}} \wedge \phi_{\text{set}} \\ \phi_{\text{basic}} &::= \top \mid o \mid s \mid \geq_n \rho . \phi_{\text{basic}} \end{aligned}$$

Algorithm 10 Definition of `TYPEOF` for λ_{SHAEL} .

global variables S , a set of shapes**function** `TYPEOF`(t, Γ)**match** t with**case** ... **then** ... ▷ existing cases
case $t_1.p$ **when** `TYPEOF`(t_1, Γ) **is** s_1 **and** s'_1 **is** `GENSHAPENAME`() **and**
`SUBTYPE`($S \cup \{(s'_1, \geq_1 p^-.s, \perp)\}, s_1, s'_1$) **then**
 $S \leftarrow S \cup \{(s'_1, \geq_1 p^-.s_1, \perp)\}$ List s'_1 **case** query q **then** $S \leftarrow S \cup \text{SHAPES}(q)$ $\{v : s_v^{\text{v} \in \text{HEAD}(q)}\}$ **case** _ **then** FAIL

A shape s' is a supertype of shape s if the following holds: Constraints for both shapes must be built according to ϕ_{set} . Furthermore, all constraints of s' must be found in the constraints for s . Subtyping in this manner is similar to subtyping for records (c. f. Section 3.2.2). Some minor improvements for special cases can be introduced. Shape s is also a subtype of shape s' if a constraint for s is a reference to s' . Likewise, s is a subtype of s' if s enforces the presence of a $-\text{type} \rightarrow$ relation which s' targets. Algorithm 11 shows the complete definition. While the algorithm can be used to type small programs such as used in Section 5.1, the approach in general is extremely limited. In particular, by removing negation, we are losing the possibility to use disjunction $\phi_1 \vee \phi_2$ in constraints. The function `LUB` for computing the least upper bound of two shapes can therefore only resort to checking whether the two shapes are in a subtype relation. The greatest lower bound on the other hand can use conjunction to construct a new shape that combines the constraints for both shapes (see Algorithm 12).

5.4. Type Elaboration

A feature that was not discussed so far is the detection of projections for which it is known that only one successor exists. For example, the shape `PersonShape` in S_1 explicitly enforces that there is exactly one $-\text{hasName} \rightarrow$ relation for each instance of `Person`. However, using the evaluation rules and type system as defined in Figure 5.4 a function `$\lambda x : \text{PersonShape} . x.\text{hasName}$` always returns a list consisting of a single graph node. Ideally, the function would not return a list, but a single graph node. Likewise, the type system needs to assign a single shape type to such projections instead of constructing a

Algorithm 11 Definition of SUBTYPE for λ_{SHACL} .

```

function SUBTYPE( $S, T, T'$ )
  match ( $T, T'$ ) with
    case ... then ...       $\triangleright$  existing cases
    case ( $s, s'$ ) when ( $s, \phi_{\text{set}}, \hat{q}$ )  $\in S$  and ( $s', \phi'_{\text{set}}, \hat{q}'$ )  $\in S$  and  $\phi_{\text{set}}$  is  $\phi_{\text{basic}}^1 \wedge \dots \wedge \phi_{\text{basic}}^n$  and
       $\phi'_{\text{set}}$  is  $\phi'_{\text{basic}}^1 \wedge \dots \wedge \phi'_{\text{basic}}^m$  and
       $\{\phi'_{\text{basic}}^1 \wedge \dots \wedge \phi'_{\text{basic}}^m\} \subseteq \{\phi_{\text{basic}}^1 \wedge \dots \wedge \phi_{\text{basic}}^n\}$  then
        true
    case ( $s, s'$ ) when ( $s, \phi_{\text{set}}, \hat{q}$ )  $\in S$  and  $\phi_{\text{set}} = \phi_{\text{basic}}^1 \wedge \dots \wedge s' \wedge \phi_{\text{basic}}^n$  then
      true
    case ( $s, s'$ ) when ( $s, \phi_{\text{set}}, \hat{q}$ )  $\in S$  and  $\phi_{\text{set}} = \phi_{\text{basic}}^1 \wedge \dots \wedge \geq_1 \text{type.class} \wedge \phi_{\text{basic}}^n$  and
      ( $s', \phi_{\text{set}}, x \leftarrow x \text{ type class}$ )  $\in S$  then
        true
    case _ then FALSE

```

Algorithm 12 Definition of LUB and GLB for λ_{SHACL} .

```

global variables
   $S$ , a set of shapes

function LUB( $T, T'$ )
  match ( $T, T'$ ) with
    case ... then ...       $\triangleright$  existing cases
    case ( $s, s'$ ) when SUBTYPE( $S, s, s'$ ) then  $s'$ 
    case ( $s, s'$ ) when SUBTYPE( $S, s', s$ ) then  $s$ 
    case _ then TOP

function GLB( $T, T'$ )
  match ( $T, T'$ ) with
    case ... then ...       $\triangleright$  existing cases
    case ( $s, s'$ ) when ( $s, \phi, \hat{q}$ )  $\in S$  and ( $s', \phi', \hat{q}'$ )  $\in S$  and  $s_{\text{GLB}}$  is GENSHAPENAME() then
       $S \leftarrow S \cup \{(s_{\text{GLB}}, \phi \wedge \phi', \perp)\}$ 
       $s_{\text{GLB}}$ 
    case _ then FAIL

```

list type. A possible solution is to rely on an *elaboration* function that transforms the program. An elaboration function typically translates a program from an external language containing *syntactic sugar* into an internal language in which these expressions are *desugared*. A programming language may offer syntactic constructs or simplified syntax for certain constructs that simplify common tasks. Such constructs are called syntactic sugar. Additional evaluation rules for these constructs can be avoided by desugaring them—that is, translating them from their extended syntax to constructs for which evaluation rules exist.

In our case, we use such an elaboration function to add an additional **head** term that directly accesses the head of the list, in cases where it is known that the list only contains a single element. As we only target projections, our elaboration function `ELABORATE` does not modify the given term in most cases (see Algorithm 13). For example, the elaboration function does not modify a λ -abstraction $\lambda x:T.t_1$,

Algorithm 13 Definition of `ELABORATE` for λ_{SHACL} .

global variables

S , a set of shapes

function `ELABORATE`(t, Γ)

match t with

case $\lambda x : T . t_1$ **then** $\lambda x : T . \text{ELABORATE}(t_1, \Gamma)$

case $\text{let } x = t_1 \text{ in } t_2$ **then** $\text{let } x = \text{ELABORATE}(t_1, \Gamma) \text{ in } \text{ELABORATE}(t_2, \Gamma)$

case $\text{head } t_1$ **then** $\text{head } \text{ELABORATE}(t_1, \Gamma)$

case $t_1.p$ **when** $\text{TYPEOF}(\Gamma, t_1)$ is s_1 **and** $\text{GENSHAPENAME}()$ is s'_1 **and**
 $\text{SUBTYPE}(S \cup \{(s'_1, =_1 p. \top, \perp)\}, s_1, s'_1)$ **then** $\text{head } t_1.p$

case $t_1.p$ **when** $\text{TYPEOF}(\Gamma, t_1)$ is s_1 **and** $\text{GENSHAPENAME}()$ is s'_1 **and**
 $\text{SUBTYPE}(S \cup \{(s'_1, \geq_1 p. \top, \perp)\}, s_1, s'_1)$ **then** $t_1.r$

case query q **then** query q

case ... **then** ... \triangleright remaining cases

but may modify its body t_1 . Likewise, a **query**-expression is not modified. Our elaboration function particularly targets role projections. If the term is a role projection $t_1.p$, we type t_1 using the `TYPEOF` function. We then check whether the resulting shape type s_1 is either a subtype of a shape having exactly one successor via p (denoted by the constraint $=_1 p. \top$) at least one successor via p (denoted by $\geq_1 p. \top$) using the `SUBTYPE` function. Depending on the result, we either add an additional **head**-term or we leave the role projection $t_1.p$ unchanged.

As an example, let us reconsider the definitions of `PersonShape` and the `getName` function as defined in Section 5.1:

$$S_1 = \{ \dots, (\text{PersonShape}, \text{hasName.StringShape}, \text{x} \leftarrow \text{x type Person}) \}$$

```

1  let getName = λ (x:PersonShape) .
2    x.hasName
3  in ...

```

Applying the elaboration function on this term adds an **head** term to the projection term `x.hasName` since `PersonShape` clearly only allows for one successor via the `-hasName→` relation:

$$\text{ELABORATE}(\emptyset, \text{let getName} = \lambda x:\text{PersonShape} . x.\text{hasName} \text{ in } \dots) =$$

$$\text{let getName} = \lambda x:\text{PersonShape} . \text{head } x.\text{hasName} \text{ in } \dots$$

5.5. Type Safety

Given the design choices of λ_{SHACL} , the language is type-safe with the routine exceptions of trying to access **head** or **tail** of empty lists. To show type safety, we first show that assigning types to queries is sound. Any shape representing a variable of a query must be sound. That is, all nodes that the variable can potentially be mapped to must conform to the shape. Using this, we can then proceed to show progress and preservation for λ_{SHACL} .

5.5.1. Soundness of Query Typing

Shape reconstruction for queries is sound if the shape constraints reconstructed for each variable evaluate to true for all possible mappings of the variable.

Definition 9 (Soundness of shape reconstruction). *Given an RDF graph G , a query q with its variables $x_i \in \text{Vars}(q)$ and the set of reconstructed shapes $S_q = \{(s_{x_i}, \phi_{x_i}, q_{s_{x_i}})_{x_i \in \text{Vars}(q)}\}$, a shape constraint is sound if there exists a faithful assignment σ such that*

$$\forall x_i \in \text{Vars}(q) : \forall \mu \in \llbracket q \rrbracket_G : \llbracket \phi_{x_i} \rrbracket^{\mu(x_i), G, \sigma} = \text{true}$$

We show that the faithful assignment σ can be constructed by assigning all shape names solely based on target nodes.

Theorem 6. *For any data graph G , a conjunctive query q and the set of shapes S_q reconstructed from q , assignment σ is constructed such that for each shape $(s, \phi_s, q_s) \in S_q$ and for each graph node $o \in \text{Nodes}(G)$:*

1. *If $o \in \llbracket q_s \rrbracket_G$, then $s \in \sigma(o)$,*

2. If $o \notin \llbracket q_s \rrbracket_G$, then $s \notin \sigma(o)$.

Such an assignment σ is faithful.

Proof. An assignment is faithful if three conditions are met. First, for all $(s, \phi_s, q_s) \in S_q$ and for all $o \in \llbracket q_s \rrbracket_G$, it must be that $s \in \sigma(o)$. This is fulfilled through the construction of σ . Furthermore, it must be true that for all $o \in \text{Nodes}(G)$:

1. if $s \in \sigma(o)$, then $\llbracket \phi_s \rrbracket^{o, G, \sigma} = \text{true}$.
2. if $s \notin \sigma(o)$, then $\llbracket \phi_s \rrbracket^{o, G, \sigma} = \text{false}$.

We show this by induction on the evaluation of $\llbracket q = (\bar{x}) \leftarrow gp \rrbracket_G$.

Q-SVAR For the query $gp = x p o'$, the reconstructed set of shapes S_q is $\{(s_x, \geq_1 p.o', \perp)\}$. Evaluation of the query returns o for which $(o, o') \in r(G)$.

1. The constraint requires all o assigned to shape s_x to have at least one successor via the relation p pointing to o' . This is true for all o since they would not be in the query result otherwise. Therefore, $s_x \in \sigma(o)$ as required by the construction of σ , does not violate faithfulness.
2. Any node $o'' \in \text{Nodes}(G)$ for which $s_x \notin \sigma(o'')$ must violate the constraint. By design of σ , any node $s_x \notin \sigma(o'')$ cannot be part of the query result. This means that they cannot have a successor via the relation p pointing to o' . Therefore, those nodes violate the constraint and σ is faithful.

Q-OVAR For the query $gp = o p x$, the reconstructed set of shapes S_q is $\{(s_x, \geq_1 p^-.o, \perp)\}$. This case is similar to case Q-SVAR.

Q-VARS For the query $gp = x_1 p x_2$, the reconstructed set of shapes S_q is $\{(s_{x_1}, \geq_1 p.s_{x_2}, \perp), (s_{x_2}, \geq_1 p^-.s_{x_1}, \perp)\}$. Evaluation of the query returns all $(o, p, o') \in G$ whereas construction of σ assigns all o to shape s_{x_1} and all o' to shape s_{x_2} .

1. The constraint requires all o to have at least one successor o' via the relation p that is assigned to the shape s_{x_2} . This is fulfilled through the construction of σ . Likewise, all o' require a predecessor via p that is assigned to s_{x_1} . Again, this must be true through the construction of σ . Therefore, the constraints evaluates to true for all o and o' respectively and the assignment σ is still faithful.
2. Any node $o'' \in \text{Nodes}(G)$ for which neither $s_{x_1} \notin \sigma(o'')$ nor $s_{x_2} \notin \sigma(o'')$ cannot have a successor or predecessor via the relation p as they would otherwise be part of the query result. Both constraints would therefore evaluate to false and σ is still faithful.

Q-CONJ For the pattern $gp = gp_1 \wedge gp_2$, sets of shapes S_{q_1} and S_{q_2} are reconstructed individually for both gp_1 and gp_2 . These are then combined into $S_q = S_{q_1} \hat{\wedge} S_{q_2}$. By induction hypothesis, σ is faithful for G and S_{q_1} and S_{q_2} individually. Evaluation of the query returns $gp_1 \bowtie gp_2$. That is, each part is evaluated individually and, for all query results μ_1 and μ_2 , the union is returned in case they are compatible. μ_1 and μ_2 are compatible if, for all variables $x \in \text{Dom}(\mu_1) \cap \text{Dom}(\mu_2)$, it holds that $\mu_1(x) = \mu_2(x)$. Therefore, for each variable x_i , there are two cases to consider:

x_i **occurring in both bodies:** $\hat{\wedge}$ takes the conjunction of the constraints for x_i in S_{q_1} and S_{q_2} .

1. By induction hypothesis, both ϕ_{i_1} from $(s_{x_i}, \phi_{i_1}, q_{i_1}) \in S_{q_1}$ and ϕ_{i_2} from $(s_{x_i}, \phi_{i_2}, q_{i_2}) \in S_{q_2}$ evaluate to true for all possible mappings of x_i . As $\hat{\wedge}$ constructs $\phi_{i_1} \wedge \phi_{i_2}$ and no negation is used in either constraint, the resulting constraint must also evaluate to true.
2. As no negation occurs in constraints of S_{q_1} and S_{q_2} , it is impossible for any nodes previously violating any constraints to fulfill the conjunction of the constraints.

x_i **only occurring in one body:** The constraint for the variable is not modified by $\hat{\wedge}$. The assignment is therefore still faithful.

Q-PROJ For the query $q = (\bar{x}) \leftarrow gp$, shapes are reconstructed for the body $gp : S_q$. This set of shapes is then used as the result for q ($q : S_q$). Results are immediate since the reconstructed set of shapes is not modified.

□

The faithful assignment σ constructed in the manner as explained above is unique. This is expected, as shape reconstruction does not use negation.

Proposition 2. *The assignment σ constructed as described above is unique.*

Proof. Assume that a different faithful assignment σ' exists. There must be at least one node o for which $\sigma(o) \neq \sigma'(o)$.

1. It is impossible that there is an s such that $s \in \sigma(o)$ and $s \notin \sigma'(o)$. σ assigns shapes based on target nodes, o must be a target node for s and σ' is not faithful.
2. It cannot be that $s \notin \sigma(o)$ and $s \in \sigma'(o)$. o must fulfill the constraint ϕ_s of shape s , otherwise σ' would not be faithful. If that is the case, then σ is not faithful. This contradicts Theorem 6.

□

Given a faithful assignment σ for a set of shapes S and the assignment σ_q for a set of shapes reconstructed from a query, the two assignments can be combined through an operator \cup which, for each graph node o , takes the union of $\sigma(o) \cup \sigma_q(o)$.

It is not possible to take the union for arbitrary faithful assignments. As a counter example, consider a set of shapes consisting of `LocalShape`, who may only know other locals and `SemilocalShape` who must know at least one node who is not a local (see Figure 5.5). Given a data graph consisting of one node `b1` which knows itself, two faithful assignments σ_1 and σ_2 exist. In assignment σ_1 , the node `b1` is

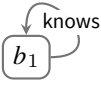
$$S_2 = \{(\text{LocalShape}, \leq_0 \text{ knows. } \neg \text{LocalShape}, \perp), \\ (\text{SemilocalShape}, \geq_1 \text{ knows. } \neg \text{LocalShape}, \perp)\}$$


Figure 5.5.: Basic example for multiple faithful assignments.

assigned to the shape `LocalShape` but not `SemilocalShape` ($\sigma_1(b_1) = \{\text{LocalShape}\}$). Likewise, in assignment σ_2 , `b1` is only assigned to `SemilocalShape` but not `LocalShape` ($\sigma_2(b_1) = \{\text{SemilocalShape}\}$). Individually, both assignments are faithful, but combining them ($\sigma_1 \cup \sigma_2$) does not yield a faithful assignment as neither constraint evaluates to true.

However, in case of σ_q for a set of shape S_q constructed from a query q , combining it with another faithful assignment σ for a set of shapes S will yield a faithful assignment again. This is because shape names of σ are unique. S cannot contain a shape (s, ϕ_s, \hat{q}) for which ϕ_s mentions a shape name s_x such that $(s_x, \phi_x, \perp) \in S_q$. Combining assignments therefore has no effect on constraint evaluation.

Proposition 3. *The assignment σ can be combined with any other assignment σ_q through an operator \cup that, for each graph node o , takes the union of σ_q and σ :*

$$\forall o \in \text{Nodes}(G) : (\sigma \cup \sigma_q)(o) = \sigma(o) \cup \sigma_q(o)$$

Proof. Shape names in σ_q are completely disjunct from shape names in σ and therefore have no effect on the evaluation of constraints. \square

5.5.2. Soundness of the Type System

Given the design choices of λ_{SHACL} , the language is type-safe. A well-typed program does not get stuck. As discussed in Section 3.2.3, the only exception to this concerns lists. We therefore show that if a program is well-typed, then the only way it can get stuck is by reaching a point where it tries to compute `head nil` or `tail nil`. As introduced in Section 2.1.3, we proceed in two steps by showing that a well-typed term is either a value or it can take a step (progress) and by showing that if that term takes a step, the result is also well-typed (preservation). We start by observing that a well-typed value of type s must be a graph node:

Lemma 3. *Let v be a well-typed value. Then one of the following must be true:*

1. *If v is a value of type `Bool`, then either $v = \text{true}$ or $v = \text{false}$.*
2. *If v is a value of type `Nat`, then v is a numerical value nv according to the grammar defined in Figure 3.1.*
3. *If v is a value of type `s`, then v is a graph node `o`.*
4. *If v is a value of type $\{l_i : T_i^{i \in 1 \dots n}\}$ then v must be a record of the form $\{l_j : t_j^{j \in 1 \dots m}\}$ with $\{l_i\} \subseteq \{l_j\}$ and $\Gamma, S \vdash t_i = T_i$ for all cases in which $i = j$.*
5. *If v is a value of type `List T`, then v is either an empty list `nil[T]` or of the form `cons v1 . . . nil[T1]` with $T_1 <: T$ and $\Gamma, S \vdash v_1 : T$.*
6. *If v is a value of type $T \rightarrow T'$, then v is a λ -abstraction `$\lambda x : T_1 . t_2$` with $T_1 <: T$ and $\Gamma, x : T_1, S \vdash t_2 : T'$.*

Given Lemma 3, we can show that a well-typed term is either a value or it can take a step.

Theorem 7 (Progress in λ_{SHACL}). *Let t be a well-typed closed term. If t is not a value, then there exists a term t' such that $t \xrightarrow{G} t'$. If there is a T such that $\Gamma, S \vdash t : T$, then t is either a value, a term containing the forms `head nil` and `tail nil`, or there is some t' with $t \xrightarrow{G} t'$.*

Proof. By induction on the derivation of $\Gamma, S \vdash t : T$. Large parts of the proof are standard cases. We therefore focus on the part specific to our language.

T-QUERY $t = \text{query } q, \quad q : S_q, \quad \Gamma, S \vdash t : \text{List } \{l_i : s_l^{i \in 1 \dots n}\}.$

Immediate since rule `E-QUERY` applies.

T-RPROJ $t = t_1.p, \quad \Gamma, S \vdash t_1 : s_1, \quad \Gamma, S \vdash t : \text{List } s'.$

By hypothesis, t_1 is either a value or it can take a step. If it can take a step, `E-RPROJ` applies. If its a value, then by Lemma 3, $t_1 = o_1$, therefore rule `E-PROJROLE` applies.

T-ADD Results follow from the induction hypothesis since rule `T-ADD` requires a term to be well-typed.

□

We can now continue to show that if a term takes a step by the evaluation rules, its type is preserved.

Theorem 8 (Preservation in λ_{SHACL}). *Let t be a term and T a type. If there is a T such that $\Gamma, S \vdash t : T$ and $t \xrightarrow{G} t'$, then $\Gamma, S \vdash t' : T$.*

Proof. By induction on the derivation of $\Gamma, S \vdash t : T$. Again, we only examine the specific cases.

T-QUERY $t = \text{query } q, \quad q : S_q, \quad \Gamma, S \vdash t : \text{List } \{l_i : s_{l_i}^{i \in 1 \dots m}\}$ with $l_i^{1 \in 1 \dots m}$ being the head of q .

If t takes a step, then it can only be through rule **E-QUERY**. By Theorem 9, all graph nodes in the evaluation results are instances of their respective concepts. Furthermore, by rules **T-RCD** and **T-LIST** the result is a list of records. The type is therefore preserved.

T-RPROJ $t = t_1.p, \quad \Gamma, S \vdash t_1 : s_1, \quad \Gamma, S \vdash t : \text{List } s' \text{ and } (s', \geq_1 r.s_1, \perp)$

There are 2 different cases by which t' can be derived:

E-PROJROLE $t' = \text{cons } \mu_1(x_1) \dots \text{cons } \mu_n(x_1) \text{ nil}$ with $\llbracket x_1 \leftarrow o_1 p x_1 \rrbracket_G = \mu_i^{i \in 1 \dots n}$.

Each node $\mu_i(x_1)$ must fulfill the constraint $\geq_1 p^- .s$ of shape s' as it would otherwise not be in the query result. There the type is preserved as $t' : \text{List } s'$.

E-RPROJ $t' = t'.p$

By hypothesis, $t_1 \xrightarrow{G} t'_1$ preserves the type. Therefore rule **T-RPROJ** applies again.

T-ADD Vacuously satisfied since there is no $t \xrightarrow{G} t'$ in this case.

□

As a direct consequence of Theorems 7 and 8, a well-typed, closed term does not get stuck during evaluation. The only exception concerns handling of lists which can get stuck if **head** or **tail** is applied to an empty list. This holds when the graph is evolving as long as the graph that results from the modification again conforms to the SHACL shapes.

5.6. Summary and Discussion

In this chapter, we have studied λ_{SHACL} —a typed λ -calculus for the Semantic Web that is built around SHACL shapes. We have shown that using SHACL shapes as types lead to a type safe language. Contrary to the type system used in Chapter 4, SHACL does not allow for modelling incomplete knowledge. An effect of this is that projection operations on nodes cannot yield empty lists. Likewise, if a SHACL shape guarantees that there is exactly one successor via a relation, then querying for this one successor yields exactly one answer. In this manner, projection now behaves similarly to accessing an attribute of an object or record in object-oriented programming languages.

However, shape containment is an open problem. The definition for subtyping so far only works in very specific cases and leaves much to be desired. For the type system to be useful, the problem of shape containment must be revisited. We therefore investigate the problem further in Chapter 6 and propose a translation of the problem to a description logic knowledge base. This, in turn, allows us to use optimized reasoner implementations such as [72] in implementations of the type system.

Shape Containment

Subtyping in λ_{SHACL} highlighted the problem of shape containment. Given a set of shapes, a shape is contained in another shape if the set of nodes conforming to the first shape are a subset of the nodes conforming to the second shape in any RDF data graph that conforms to the set of shapes. While some preliminary work for other RDF validation languages exists (c. f. [87]), containment in SHACL has not been considered so far as it is not used in the validation of RDF graphs. We investigate a translation of the containment problem into a DL concept subsumption problem. For a subset of SHACL that does not use inverse roles, the result is a sound and complete approach for deciding containment. For SHACL as defined in this thesis, the translation provides a sound but incomplete approach. This chapter is based on [67].

6.1. Problem Description

Containment problems are used in a number of different settings such as query optimization [14, 15, 40] as well as data exchange or graph summaries [87]. We investigate the problem of *shape containment*: That is, given a set of shapes S and two shapes $s, s' \in \text{Names}(S)$, is the set of nodes that conform to s contained in the set of nodes conforming to s' . As an example, let us consider the following set of shapes:

$$\begin{aligned} S_3 = \{ & (\text{StudentShape}, \geq_1 \text{ studiesAt} . \top \wedge \forall \text{ studiesAt} . \text{UniversityShape} \\ & \wedge \geq_1 \text{ type} . \text{Person}, x \leftarrow x \text{ type Student}), \\ & (\text{PersonShape}, =_1 \text{ hasName} . \top, x \leftarrow x \text{ type Person}), \\ & (\text{UniversityShape}, \geq_1 \text{ locatedIn} . \top \wedge \geq_1 \text{ type} . \text{University}, \perp), \\ & (\text{OnlineUniversityShape}, \geq_1 \text{ studiesAt}^- . \text{PersonShape} \wedge \geq_1 \text{ type} . \text{University}, \perp) \} \end{aligned}$$

The shape $\boxed{\text{StudentShape}}$ targets all instances of $\boxed{\text{Student}}$. It enforces the presence of at least one $-\text{studiesAt} \rightarrow$ relation and that everything reachable via that relation conforms to the $\boxed{\text{UniversityShape}}$ shape. Furthermore, nodes conforming to $\boxed{\text{StudentShape}}$ must be instances of $\boxed{\text{Person}}$. $\boxed{\text{PersonShape}}$ targets all instances of $\boxed{\text{Person}}$ and enforces the presence of exactly one $-\text{hasName} \rightarrow$ relation. $\boxed{\text{UniversityShape}}$ enforces

that nodes are an instance of `University` and that there is a `-locatedIn→` relation. However, there are no target nodes for `UniversityShape`. Lastly, we added a new shape named `OnlineUniversityShape` which again has no target nodes. For the sake of the example, its constraints enforce the presence of an incoming `-studiesAt→` relation from a node conforming to `PersonShape` as well as being an instance of `University`. However, it does not have a `-locatedIn→` relation.

The shape `StudentShape` is, for example, contained in the shape `PersonShape`. All nodes conforming to `StudentShape` are required to be instances of `Person` and `PersonShape` targets all instances of `Person`. Subsequently, all nodes conforming to `StudentShape` must also conform to `PersonShape`. On the other hand, the shape `OnlineUniversityShape` is not contained in the shape `UniversityShape`. This can be shown through a counterexample in the form of an RDF graph and a faithful assignment in which a node is assigned to `OnlineUniversityShape` but not to `UniversityShape` (see Figure 6.1). Formally, a shape s being contained in

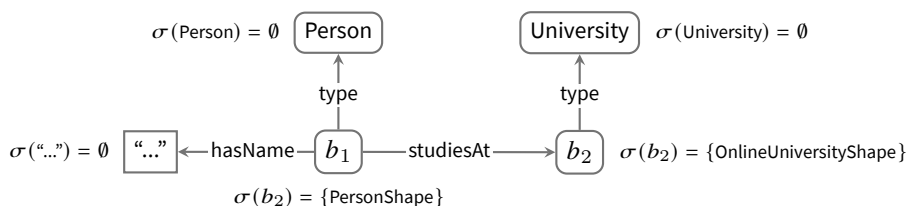


Figure 6.1.: Counterexample that proves that `OnlineUniversityShape` is not contained in `UniversityShape`.

another shape s' with respect to a set of shapes S means that for all possible faithful assignments over all possible RDF data graphs, being assigned to s implies also being assigned to s' .

Definition 10 (Shape Containment). Let S be a set of shapes. Furthermore, let \mathcal{G} be the set of all possible RDF graphs, let G be an individual RDF graph and let $\text{Faith}(G, S)$ be the set of faithful assignments for G and S . The shape s is contained in shape s' if:

$$\forall G \in \mathcal{G} : \forall \sigma \in \text{Faith}(G, S) : \forall o \in \text{Nodes}(G) : s \in \sigma(o) \Rightarrow s' \in \sigma(o) \text{ with } s, s' \in \text{Names}(S)$$

We use the notation $s <:_S s'$ to indicate that shape s is contained in the shape s' .

Tableau-based approaches constitute the most widely used technique for solving decision problems in description logics [19]. For subsumption in particular, algorithms try to construct counterexamples. When trying to prove that a concept expressions `Person` subsumes a concept expression `Student` in a knowledge base K , a reasoner attempts to construct a model of K in which the concept expression `Student \sqcap \neg Person` is satisfiable. This either results in a valid model or in the discovery of a contradiction that proves that no model can exist. If a model exist, then it acts as a counterexample. We leverage this approach by mapping a set of SHACL shapes to a description logic knowledge base. This then allows for the decision of shape containment using description logic reasoners.

We proceed by syntactically mapping sets of shapes into description logic knowledge bases. We then show the equivalence of faithful assignments for SHACL shapes and finite models for description logic knowledge bases. We then continue to show that the problem of concept subsumption in this description logic knowledge base is equivalent to shape containment if SHACL is restricted to a subset of the available syntactic constructs.

6.2. From SHACL to Description Logic

We start by syntactically mapping sets of shapes into description logic knowledge bases. That is, we define a function τ_{shapes} that maps a set of shapes S to a description logic knowledge base K^S using four functions: First, τ_{name} maps shape names, RDF classes as well as properties and graph nodes onto atomic concept names, atomic property names and object names. Second, τ_{role} maps SHACL path expressions to DL role expressions. Third, τ_{constr} maps constraints to concept expressions. Fourth, τ_{target} maps queries for target nodes to concept expressions. The functions map the set of shapes such that $s <_S s'$ is true if $K^S \models \tau_{\text{name}}(s) \sqsubseteq \tau_{\text{name}}(s')$. For this, we show that finite models of K^S are equivalent to faithful assignments and vice versa.

Syntactic mapping First, we provide details on the syntactic mapping of sets of shapes S onto a description logic knowledge base K^S . To reiterate the most important definitions in Section 2.2.4: A set of shapes S consists of triples (s, ϕ, \hat{q}) where $s \in \text{Names}(S)$ and the constraint ϕ is built according to the following grammar with r being either a normal property p or the inverse of a path p^- :

$$\phi ::= \top \mid s \mid o \mid \phi \wedge \phi \mid \neg \phi \mid \geq_n \rho.\phi$$

Furthermore, a query for target nodes \hat{q} takes one the following forms: (1) It can select no nodes at all (\perp). (2) It can enumerate the target nodes (\bar{o}). It can define target nodes based on a *concept* ($x_1 \leftarrow x_1 \text{ type } \textit{concept}$). (4) It can select nodes based on a *property* ($x_1 \leftarrow x_1 \textit{property } x_2$ and $x_2 \leftarrow x_1 \textit{property } x_2$).

Mapping these elements requires a sufficiently expressive target description logic. In this description logic, we must be able to express negation and conjunction which requires the DL \mathcal{ALC} . Furthermore, we must be able to use individual graph nodes as concept expressions, which requires nominals (\mathcal{O}), inverse path expressions (\mathcal{I}) and qualified number restrictions (\mathcal{Q}). The description logic that corresponds to SHACL as used in this thesis is therefore \mathcal{ALCOIQ} . The function τ_{shapes} maps a set of shapes S into a knowledge base K^S by mapping constraints and target node queries of each shape using the functions τ_{role} , τ_{constr} and τ_{target} . All those functions rely on τ_{name} which maps atomic elements used in SHACL to atomic elements of a DL knowledge base.

Definition 11 (Mapping atomic elements). *The function τ_{name} is an injective function mapping shape names and RDF classes onto atomic concept names, graph nodes onto object names as well as properties onto atomic property names.*

Definition 12 (Mapping path expressions to description logic roles atomic elements). *The path mapping function $\tau_{\text{role}} : \mathcal{P} \rightarrow \mathcal{R}$ is defined as follows:*

$$\begin{aligned}\tau_{\text{role}}(p) &= \tau_{\text{name}}(p) \\ \tau_{\text{role}}(\rho^-) &= \tau_{\text{role}}(\rho)^-\end{aligned}$$

Definition 13 (Mapping constraints to DL concept expressions). *The function $\tau_{\text{constr}} : \Phi \rightarrow C$, which takes a constraint ϕ and returns a DL concept expression C is then defined as follows:*

$$\begin{aligned}\tau_{\text{constr}}(\top) &= \top \\ \tau_{\text{constr}}(s) &= \tau_{\text{name}}(s) \\ \tau_{\text{constr}}(o) &= \{\tau_{\text{name}}(o)\} \\ \tau_{\text{constr}}(\phi_1 \wedge \phi_2) &= \tau_{\text{constr}}(\phi_1) \sqcap \tau_{\text{constr}}(\phi_2) \\ \tau_{\text{constr}}(\neg\phi) &= \neg\tau_{\text{constr}}(\phi) \\ \tau_{\text{constr}}(\geq_n \rho.\phi) &= \geq_n \tau_{\text{role}}(\rho).\tau_{\text{constr}}(\phi)\end{aligned}$$

Definition 14 (Mapping target node queries to DL concept expressions). *Concepts that are used as target nodes for shapes are represented by atomic concepts A_{concept} . The function $\tau_{\text{target}} : \hat{Q} \rightarrow C$, which takes a query for target nodes \hat{q} and returns a concept expression C , is defined as follows:*

$$\begin{aligned}\tau_{\text{target}}(\perp) &= \perp \\ \tau_{\text{target}}(\{\bar{o}\}) &= \{\tau_{\text{name}}(\bar{o})\} \\ \tau_{\text{target}}(x_1 \leftarrow x_1 \text{ type concept}) &= \tau_{\text{name}}(\text{concept}) \\ \tau_{\text{target}}(x_1 \leftarrow x_1 \text{ property } x_2) &= \exists \tau_{\text{name}}(\text{property}).\top \\ \tau_{\text{target}}(x_2 \leftarrow x_1 \text{ property } x_2) &= \exists \tau_{\text{name}}(\text{property})^-. \top\end{aligned}$$

The functions are defined such that the translation retains the original meaning of the expressions. For constraints, this means that graph nodes which evaluate to true for the constraint should be in the interpretation of the concept expression. The constraint $\geq_1 \text{ studiesAt}.\top$ evaluates to true for all graph nodes that have at least one $\text{--studiesAt--}\rightarrow$ relation. Likewise, the interpretation of its translation $\geq_1 \text{ studiesAt}.\top$ contains all graph nodes that have at least one $\text{--studiesAt--}\rightarrow$ relation. For target node queries, all nodes that are returned by the evaluation of the query should be in the interpretation of the concept expression. The target node query $x_2 \leftarrow x_1 \text{ studiesAt } x_2$ returns all graph nodes that have an incoming $\text{--studiesAt--}\rightarrow$ relation. Likewise, the interpretation of its translation $\exists \text{ studiesAt}^-. \top$ contains all graph nodes with an incoming $\text{--studiesAt--}\rightarrow$ relation.

Lastly, the function τ_{shapes} takes a set of shapes and translates it into a knowledge base.

Definition 15 (Mapping sets of shapes to DL axioms). *The function $\tau_{\text{shapes}} : \mathcal{S} \rightarrow \mathcal{K}$, which takes a set*

of shapes S and returns a set of DL axioms K , is defined as follows:

$$\tau_{\text{shapes}}(S) = \bigcup_{(s, \phi, \hat{q}) \in S} \{ \tau_{\text{target}}(\hat{q}) \sqsubseteq \tau_{\text{name}}(s), \tau_{\text{name}}(s) \equiv \tau_{\text{constr}}(\phi) \}$$

Again, the function τ_{shapes} is defined such that it preserves the original meaning of faithful assignments. To reiterate, an assignment is faithful if two conditions hold: For one, all target nodes of a shape must be assigned to the shape. Therefore, the first axiom $\tau_{\text{target}}(\hat{q}) \sqsubseteq \tau_{\text{name}}(s)$ states that the concept expression representing the query for target nodes is subsumed by the concept representing the shape. Second, an assignment is only faithful if all nodes for which the constraint evaluates to true are assigned to the shape and vice versa. This is expressed through the second axiom $\tau_{\text{name}}(s) \equiv \tau_{\text{constr}}(\phi)$, enforcing equivalence between the set of nodes that are assigned to s and the nodes for which the constraint evaluates to true. Let us exemplify this translation using the following set of shapes:

$$S_2 = \left\{ \begin{array}{l} (\text{StudentShape}, \quad \geq_1 \text{studiesAt.}\top \wedge \forall \text{studiesAt.UniversityShape}, \quad x_1 \leftarrow x_1 \text{ type Student}), \\ (\text{UniversityShape}, \quad \geq_1 \text{locatedIn.}\top, \quad \perp) \end{array} \right\}$$

The shape StudentShape is translated as follows:

$$\begin{aligned} \tau_{\text{constr}}(\geq_1 \text{studiesAt.}\top \wedge \forall \text{studiesAt.UniversityShape}) &= \geq 1 \text{studiesAt.}\top \sqcap \\ &\quad \forall \text{studiesAt.UniversityShape} \\ \tau_{\text{target}}(x_1 \leftarrow x_1 \text{ type Student}) &= \text{Student} \end{aligned}$$

Likewise, the shape UniversityShape is translated as follows:

$$\begin{aligned} \tau_{\text{constr}}(\geq_1 \text{locatedIn.}\top) &= \geq 1 \text{locatedIn.}\top \\ \tau_{\text{target}}(\perp) &= \perp \end{aligned}$$

The final translation then looks as follows:

$$\begin{aligned} \tau_{\text{shapes}}(S_2) = \{ &\text{Student} \sqsubseteq \text{StudentShape}, \\ &\geq 1 \text{studiesAt.}\top \sqcap \forall \text{studiesAt.UniversityShape} \equiv \text{StudentShape}, \\ &\perp \sqsubseteq \text{UniversityShape}, \\ &\geq 1 \text{locatedIn.}\top \equiv \text{UniversityShape} \end{aligned} \}$$

Equivalence of faithful assignments and models Given our translation, we now show that the notion of faithful assignments of SHACL and models in description logics coincide. In particular, this is the case for *finite models* of the knowledge base K^S .

Definition 16 (Finite model and finitely satisfiable). *Let K be a knowledge base and $I \in \text{Mod}(K)$ an model of K . The model I is finite, if its universe Δ^I is finite [28]. A concept expression C is finitely satisfiable in I if I is finite and $C^I \neq \emptyset$.*

Given a finite model I of a knowledge base K^S that is constructed from a set of shapes S , it is possible to construct an RDF data graph G^I and an assignment σ^I such that σ^I is faithful with respect to S and G^I . As RDF graphs are finite sets of triples, it is impossible to construct an RDF graph from an infinitely large model of K^S . Given an RDF data graph G and an assignment σ that is faithful with respect to some set of shapes S and G , it is also possible to construct an interpretation $I^{G,\sigma}$ that is a finite model for the knowledge base K^S .

We start by constructing the interpretation $I^{G,\sigma}$ for the knowledge base K^S .

Definition 17 (Construction of the interpretation $I^{G,\sigma}$). *Let S be a set of shapes, G an RDF data graph and σ an assignment that is faithful with respect to S and G . Furthermore, let τ_{node} be the inverse of the function τ_{name} . The interpretation $I^{G,\sigma}$ for a knowledge base $\tau_{\text{shapes}}(S) = K^S$ is constructed as follows:*

1. All objects are interpreted as themselves: $\forall o \in N_O : o^I = o$.
2. A pair of objects is contained in the interpretation of a relation if the two objects are connected in the RDF data graph: $\forall p \in N_P : \forall o, o' \in N_O : (o^I, o'^I) \in p^I$ iff $(\tau_{\text{node}}(o), p, \tau_{\text{node}}(o')) \in G$.
3. Objects are in the interpretation of a concept if this concept is a class used in the RDF data graph and the object is an instance of this class according to the graph:

$$\forall A \in N_A : \forall o \in N_O : o^I \in A^I$$
 iff $(\tau_{\text{node}}(o), \text{type}, \tau_{\text{node}}(A)) \in G$.
4. Objects are in the interpretation of a concept if the concept is a shape name and the assignment σ assigns the shape to the object: $\forall A \in N_A : \forall o \in N_O : o^I \in A^I$ iff $A \in \sigma(o)$.

An interpretation $I^{G,\sigma}$ constructed in this manner is then a model of the knowledge base $\tau_{\text{shapes}}(S) = K^S$.

Theorem 9. *Let S be a set of shapes, G an RDF data graph and σ an assignment that is faithful with respect to S and G . Furthermore, let $\tau_{\text{shapes}}(S) = K^S$ be a knowledge base. The interpretation $I^{G,\sigma}$ constructed as described above is a model of K^S ($I^{G,\sigma} \models K^S$).*

Proof. $I^{G,\sigma}$ is a model of K^S iff $\forall \text{stat} \in K^S : I^{G,\sigma} \models \text{stat}$. For each shape $(s, \phi, \hat{q}) \in S$, there are two axioms in K^S . First, the axiom $\tau_{\text{target}}(\hat{q}) \sqsubseteq \tau_{\text{name}}(s)$. Second, the axiom $\tau_{\text{name}}(s) \equiv \tau_{\text{constr}}(\phi)$. $I^{G,\sigma}$ must satisfy both axioms. We start by showing that $\tau_{\text{target}}(\hat{q}) \sqsubseteq \tau_{\text{name}}(s)$ is satisfied in $I^{G,\sigma}$ by examining each case of τ_{target} individually:

$\tau_{\text{target}}(\perp) = \perp$ Vacuously satisfied as \perp is a subset of every concept expression.

$\tau_{\text{target}}(\{\bar{o}\}) = \{\tau_{\text{name}}(\bar{o})\}$ Target nodes consist of an enumeration of nodes. σ is only faithful if the shape s is assigned to all those nodes. Likewise, $\{\tau_{\text{name}}(\bar{o})\}$ constitutes a concept expression that is an enumeration of graph nodes. As all nodes that are assigned to s in σ are also in the interpretation $\tau_{\text{name}}(s)^{I^{G,\sigma}}$ of $\tau_{\text{name}}(s)$, the axiom $\{\bar{o}\} \sqsubseteq \tau_{\text{name}}(s)$ must be true in $I^{G,\sigma}$.

$\tau_{\text{target}}(x_1 \leftarrow x_1 \text{ type } \textit{concept}) = A_{\textit{concept}}$ The assignment σ is only faithful if the shape s is assigned to all instances of *concept*. Due to the construction of $I^{G,\sigma}$, all instances of *concept* are in the interpretation $\tau_{\textit{name}}(s)^{I^{G,\sigma}}$ of $\tau_{\textit{name}}(s)$. Subsequently, $A_{\textit{concept}} \sqsubseteq \tau_{\textit{name}}(s)$ must be true in $I^{G,\sigma}$.

$\tau_{\text{target}}(x_1 \leftarrow x_1 \text{ property } x_2) = \exists \tau_{\textit{name}}(\textit{property}).\top$ The assignment σ is faithful if shape s is assigned to all nodes that have the given *property*. Since the interpretation $I^{G,\sigma}$ is constructed using σ , all nodes having that property must be in the interpretation $\tau_{\textit{name}}(s)^{I^{G,\sigma}}$ of $\tau_{\textit{name}}(s)$. Subsequently, $\exists \tau_{\textit{name}}(\textit{property}).\top \sqsubseteq \tau_{\textit{name}}(s)$ must be true in $I^{G,\sigma}$.

$\tau_{\text{target}}(x_2 \leftarrow x_1 \text{ property } x_2) = \exists \tau_{\textit{name}}(\textit{property})^{\neg}.\top$ The assignment σ is faithful if shape s is assigned to all nodes that have the given incoming *property* relation. Due to the construction of $I^{G,\sigma}$ (c. f. Definition 17), all nodes that have an incoming relation via the property are in the interpretation of $\tau_{\textit{name}}(s)$. Subsequently, the axiom $\exists \tau_{\textit{name}}(\textit{property})^{\neg}.\top \sqsubseteq \tau_{\textit{name}}(s)$ must be satisfied in $I^{G,\sigma}$.

We continue by showing that $\tau_{\text{constr}}(\phi) \equiv \tau_{\textit{name}}(s)$ is true in $I^{G,\sigma}$ via induction over τ_{constr} :

$\tau_{\text{constr}}(\top) = \top$ If $\phi = \top$, then $\llbracket \top \rrbracket^{o,G,\sigma}$ evaluates to true for all nodes. Therefore the shape s is assigned to all nodes and subsequently the concept $\tau_{\textit{name}}(s)^{I^{G,\sigma}}$ contains all nodes due to the construction of $I^{G,\sigma}$. This is equivalent to the concept \top .

$\tau_{\text{constr}}(s') = \tau_{\textit{name}}(s')$ If $\phi = s'$, then $\llbracket s' \rrbracket^{o,G,\sigma}$ evaluates to true if $s' \in \sigma(o)$. Due to the construction of $I^{G,\sigma}$, all nodes for which this is true must also be in the interpretation $\tau_{\textit{name}}(s)^{I^{G,\sigma}}$ of $\tau_{\textit{name}}(s)$. Subsequently, the axiom $\tau_{\textit{name}}(s') \equiv \tau_{\textit{name}}(s)$ must be true in $I^{G,\sigma}$.

$\tau_{\text{constr}}(o) = \{\tau_{\textit{name}}(o)\}$ If $\phi = o$, then $\llbracket o \rrbracket^{o',G,\sigma}$ evaluates to true if $o = o'$. Therefore, the shape s is assigned to this node. Due to the construction of $I^{G,\sigma}$, $\tau_{\textit{name}}(o)$ is the only one in the interpretation of $\tau_{\textit{name}}(s)^{I^{G,\sigma}}$. The axiom $\{\tau_{\textit{name}}(o)\} \equiv \tau_{\textit{name}}(s)$ must therefore be true in $I^{G,\sigma}$.

$\tau_{\text{constr}}(\phi_1 \wedge \phi_2) = \tau_{\text{constr}}(\phi_1) \sqcap \tau_{\text{constr}}(\phi_2)$ Evaluation of the constraint $\llbracket \phi_1 \wedge \phi_2 \rrbracket^{o,G,\sigma}$ evaluates to true for those nodes for which both ϕ_1 and ϕ_2 evaluate to true. By induction hypothesis, $\tau_{\text{constr}}(\phi_1) = C$ is a concept expression that is equivalent to the set of nodes for which ϕ_1 evaluates to true. Likewise for $\tau_{\text{constr}}(\phi_2) = D$. The set of nodes for which both ϕ_1 and ϕ_2 evaluate to true must therefore be the intersection of $C \sqcap D$. Due to the construction of $I^{G,\sigma}$, those nodes must also be in the interpretation $\tau_{\textit{name}}(s)^{I^{G,\sigma}}$ of $\tau_{\textit{name}}(s)$. The axiom must therefore be true.

$\tau_{\text{constr}}(\neg\phi_1) = \neg\tau_{\text{constr}}(\phi_1)$ By hypothesis, $\tau_{\text{constr}}(\phi_1) = C$ is a concept expression that is equivalent to the set of nodes for which ϕ_1 evaluates to true. Evaluation of the constraint $\llbracket \neg\phi_1 \rrbracket^{o,G,\sigma}$ evaluates to true for those nodes in which ϕ_1 evaluates to false. Since those nodes are assigned to s in σ , the interpretation $\tau_{\text{name}}(s)^{I^{G,\sigma}}$ of $\tau_{\text{name}}(s)$ must also be those nodes. This is equivalent to the interpretation of $\neg C$. The axiom $\neg C \equiv \tau_{\text{name}}(s)$ must therefore be satisfied in $I^{G,\sigma}$.

$\tau_{\text{constr}}(\geq_n \rho.\phi_1) = \geq_n \tau_{\text{role}}(\rho).\tau_{\text{constr}}(\phi_1)$ By hypothesis, $\tau_{\text{constr}}(\phi_1) = C$ is a concept expression that represents the set of graph nodes for which ϕ_1 evaluates to true. $\llbracket \geq_n \rho.\phi_1 \rrbracket^{o,G,\sigma}$ evaluates to true for those nodes that have at least n successors via ρ and for which ϕ_1 evaluates to true. Those nodes must also be in the interpretation $\tau_{\text{name}}(s)^{I^{G,\sigma}}$ of $\tau_{\text{name}}(s)$. Due to the construction of $I^{G,\sigma}$, all graph nodes having n successor via ρ in G must also have n successors in $I^{G,\sigma}$. Subsequently, the axiom $\geq_n \tau_{\text{role}}(\rho).\tau_{\text{constr}}(\phi_1) \equiv \tau_{\text{name}}(s)$ must be true in $I^{G,\sigma}$.

□

Furthermore, given a finite model I of a knowledge base $\tau_{\text{shapes}}(S) = K^S$ built from a set of shapes S , I can be transformed into an RDF graph G^I and an assignment σ^I such that σ^I is faithful with respect to S and G^I . We construct G^I and σ in the following manner:

Definition 18 (Construction of G^I and σ^I). *Let S be a set of shapes and $\tau_{\text{shapes}}(S) = K^S$ a knowledge base constructed from S and let τ_{node} be the inverse of the function τ_{name} . Furthermore, let $I \in \text{Mod}(K^S)$ be a finite model of K^S . The RDF graph G^I and the assignment σ^I can then be constructed as follows:*

1. *The syntactic elements for the objects in the interpretations of all relations are interpreted as relations between graph nodes in the RDF graph: $\forall p \in N_P : (o^I, o'^I) \in p^I \Rightarrow (\tau_{\text{node}}(o), p, \tau_{\text{node}}(o')) \in G^I$.*
2. *The interpretations of all concepts that are not shape names are triples indicating an instance in the RDF graph: $\forall A \in N_A : (o^I \in A^I \wedge A \notin \text{Names}(S)) \Rightarrow (\tau_{\text{node}}(o), \text{type}, \tau_{\text{node}}(A)) \in G$.*
3. *The interpretations of all concept names that are shape names are used to construct the assignment σ : $\forall A \in N_A : (o^I \in A^I \wedge A \in \text{Names}(S)) \Rightarrow \tau_{\text{node}}(A) \in \sigma^I(\tau_{\text{node}}(o))$.*

An assignment σ^I constructed in this manner is faithful with respect to the constructed RDF graph G^I and the set of shapes S .

Theorem 10. *Let S be a set of shapes and $\tau_{\text{shapes}}(S) = K^S$ be a knowledge base constructed from S . Furthermore, let $I \in \text{Mod}(K^S)$ be a finite model for K^S . The assignment σ^I constructed as described in Definition 18 is faithful with respect to S and G^I .*

Proof. σ^I is faithful with respect to S and G^I if two conditions hold:

1. Each shape is assigned to all of its target nodes.
2. If a shape is assigned to a node, then the constraint evaluates to true. Likewise, if the constraint evaluates to true for a node, then the shape is assigned to the node.

The knowledge base $\tau_{\text{shapes}}(S) = K^S$ contains two axioms of the form $\tau_{\text{target}}(\hat{q}) \sqsubseteq \tau_{\text{name}}(s)$ and $\tau_{\text{constr}}(\phi) \equiv \tau_{\text{name}}(s)$ for each (s, ϕ, \hat{q}) which $I \in \text{Mod}(K^S)$ satisfies. We proceed by examining each case of τ_{target} individually:

$\tau_{\text{target}}(\perp) = \perp$ Vacuously satisfied as no target nodes exist.

$\tau_{\text{target}}(\{\bar{o}\}) = \{\overline{\tau_{\text{name}}(o)}\}$ The target nodes are an enumeration of nodes $\{\bar{o}\}$. I is only a model if $\{\tau_{\text{name}}(\bar{o})\}^I \subseteq \tau_{\text{name}}(s)^I$ is true. Due to the construction of G^I , all nodes \bar{o} must exist in G^I . Due to the construction of σ^I , the shape s is assigned to all nodes in $\tau_{\text{name}}(s)^I$. As such, the shape is assigned to all of its target nodes.

$\tau_{\text{target}}(x_1 \leftarrow x_1 \text{ type concept}) = A_{\text{concept}}$ Target nodes are instances of a concept. I is only a model if $A_{\text{concept}}^I \subseteq \tau_{\text{name}}(s)^I$ is true. Due to the construction of G^I , the concept A_{concept} and its instances A_{concept}^I must exist in G^I . Due to the construction of σ^I , s is assigned to all nodes in A_{concept}^I . As such, the shape is assigned to all of its target nodes.

$\tau_{\text{target}}(x_1 \leftarrow x_1 \text{ property } x_2) = \exists \tau_{\text{name}}(\text{property}).\top$ Target nodes are all subjects of a property. I is only a model if $\exists \tau_{\text{name}}(\text{property}).\top^I \subseteq \tau_{\text{name}}(s)^I$ is true. Due to the construction of G^I , all nodes in $\exists \tau_{\text{name}}(\text{property}).\top^I$ must exist in G^I and have the property. Due to the construction of σ^I , the shape s is assigned to all nodes in $\exists \tau_{\text{name}}(\text{property}).\top^I$. As such, the shape is assigned to all of its target nodes.

$\tau_{\text{target}}(x_2 \leftarrow x_1 \text{ property } x_2) = \exists \tau_{\text{name}}(\text{property}).\top$ Target nodes are objects of a property. The case is similar to the previous case.

We continue by examining each case of τ_{constr} individually:

$\tau_{\text{constr}}(\top) = \top$ The constraint evaluates to true for all nodes. If I is a model, then $\top^I = \tau_{\text{name}}(s)^I$ is true. Due to the construction of σ^I , s is assigned to all nodes. As such, the shape is assigned to all nodes for which the constraint evaluates to true.

$\tau_{\text{constr}}(s') = A_{s'}$ The constraint evaluates to true for all nodes o for which $s' \in \sigma^I(o)$. Since I is a model, $A_{s'}^I \equiv \tau_{\text{name}}(s)^I$ must be true. Due to the construction of σ^I , s is assigned to all nodes which are also assigned to s' . As such, the shape is assigned to all nodes for which the constraint evaluates to true.

$\tau_{\text{constr}}(o) = \{\tau_{\text{name}}(o)\}$ The constraint evaluates only for the node o to true. Since I is a model, $\{o\}^I \equiv \tau_{\text{name}}(s)^I$ must be true. Due to the construction of σ^I , s is only assigned to the node o .

$\tau_{\text{constr}}(\phi_1 \wedge \phi_2) = \tau_{\text{constr}}(\phi_1) \sqcap \tau_{\text{constr}}(\phi_2)$ The constraint evaluates to true for all nodes for which ϕ_1 and ϕ_2 evaluate to true. By hypothesis, $\tau_{\text{constr}}(\phi_1) = C$ and $\tau_{\text{constr}}(\phi_2) = D$ represent the set of nodes for which ϕ_1 and ϕ_2 evaluate to true, respectively. Furthermore, $(C \sqcap D)^I \equiv \tau_{\text{name}}(s)^I$ is true if I is a model. Due to the construction of σ^I , the shape s is assigned to all nodes in $(C \sqcap D)^I$. The shape is therefore assigned to all nodes for which the constraint evaluates to true.

$\tau_{\text{constr}}(\neg\phi_1) = \neg\tau_{\text{constr}}(\phi_1)$ The constraint evaluates to true for all nodes for which ϕ_1 does not evaluate to true. By hypothesis, $\tau_{\text{constr}}(\phi_1) = C$ represent the set of nodes for which ϕ_1 evaluates to true. Furthermore, since I is a model, $(\neg C)^I = \tau_{\text{name}}(s)^I$ must be true. Due to the construction of σ^I , the shape s is assigned to all nodes for which ϕ_1 evaluates to false. The shape is therefore assigned to all nodes for which the constraint evaluates to true.

$\tau_{\text{constr}}(\geq_n \rho \cdot \phi_1) = \geq_n \tau_{\text{role}}(\rho) \cdot \tau_{\text{constr}}(\phi_1)$ The constraint evaluates to true for all nodes that have n successors via ρ for which the constraint ϕ_1 evaluates to true. By hypothesis, $\tau_{\text{constr}}(\phi_1) = C$ is the set of nodes for which ϕ_1 evaluates to true. Furthermore, since I is a model, $(\geq_n \tau_{\text{role}}(\rho) \cdot \tau_{\text{constr}}(C))^I = \tau_{\text{name}}(s)^I$. Due to the construction of σ^I , the shape s is assigned to all nodes that have n successors and that are in the interpretation of C^I . The shape is therefore assigned to all nodes for which the constraint evaluates to true.

□

Given the translation rules and semantic equivalence between models of a description logic and assignments for SHACL shapes, we can leverage description logics for deciding shape containment. Assume a set of shapes S containing definitions for two shapes s and s' . Those shapes are represented by atomic concepts of the same name in the knowledge base K^S . Deciding whether the shape s is contained in the shape s' is equivalent to deciding concept subsumption between s and s' in K^S using finite model reasoning.

Theorem 11 (Shape containment and concept subsumption). *Let S be a set of shapes and $\tau_{\text{shapes}}(S) = K^S$ a knowledge base constructed from S . Then it holds that:*

$$s <_S s' \text{ with } s, s' \in \text{Names}(S) \Leftrightarrow K^S \models_{\text{finite}} \tau_{\text{name}}(s) \sqsubseteq \tau_{\text{name}}(s')$$

Proof. Intuitively, the two problems are equivalent because any counterexample for one side of the equivalence relation could always be translated into a counterexample for the other side. We proceed

by showing both directions of the equivalence relation. If $K^S \not\models_{\text{finite}} \tau_{\text{name}}(s) \sqsubseteq \tau_{\text{name}}(s')$, then there is a finite model of K^S in which $\tau_{\text{name}}(s) \sqsubseteq \tau_{\text{name}}(s')$ is not true. Instead, there must be a model in which the concept expression $\tau_{\text{name}}(s) \sqcap \neg \tau_{\text{name}}(s')$ is true. Using Definition 18, this model can be translated into an RDF graph and an assignment (c. f. Theorem 10) that acts as a counterexample for s being contained in s' . If $K^S \models_{\text{finite}} \tau_{\text{name}}(s) \sqsubseteq \tau_{\text{name}}(s')$, then there is no finite model in which $\tau_{\text{name}}(s) \sqsubseteq \tau_{\text{name}}(s')$ is not true. Subsequently, there cannot be an RDF graph and an assignment that acts as a counterexample to s being contained in s' , because this could be translated into a finite model of K^S using Definition 17 (c. f. Theorem 9) and we know that no such model exists. \square

As an example, let us reconsider the set of shapes S_3 as defined in Section 6.1. The knowledge base $\tau_{\text{shapes}}(S_3) = K^{S_3}$ is constructed as follows:

$$\begin{aligned}
K^{S_3} = \{ & \text{Student} \sqsubseteq \text{StudentShape}, \\
& \geq 1 \text{studiesAt} . \top \sqcap \forall \text{studiesAt} . \text{UniversityShape} \sqcap \text{Person} \equiv \text{StudentShape}, \\
& \text{Person} \sqsubseteq \text{PersonShape}, \\
& = 1 \text{hasName} . \top \equiv \text{PersonShape}, \\
& \perp \sqsubseteq \text{UniversityShape}, \\
& \geq 1 \text{locatedIn} . \top \sqcap \text{University} \equiv \text{UniversityShape}, \\
& \perp \sqsubseteq \text{OnlineUniversityShape}, \\
& \geq 1 \text{studiesAt}^- . \text{Person} \sqcap \text{University} \equiv \text{OnlineUniversityShape} \quad \}
\end{aligned}$$

Again, it holds that $K^{S_3} \not\models \text{OnlineUniversityShape} \sqsubseteq \text{UniversityShape}$ as there is a counterexample that can be constructed from the RDF graph and faithful assignment as shown in Figure 6.1. The interpretation $I_1 = (\cdot^I, \Delta^I)$ can be defined as follows: The universe is constructed from the graph nodes $\Delta^{I_1} = \{\overline{b_1}, \overline{b_2}, \dots\}$. Interpretations of concepts and relations are then as follows: $\text{Person}^{I_1} = \{\overline{b_1}\}$, $\text{University}^{I_1} = \{\overline{b_2}\}$, $\text{hasName}^{I_1} = \{(\overline{b_1}, \dots)\}$ and $\text{studiesAt}^{I_1} = \{(\overline{b_1}, \overline{b_2})\}$. In this interpretation, it holds that $I_1 \models \overline{b_2} : \text{OnlineUniversityShape} \sqcap \neg \text{UniversityShape}$.

For shapes belonging to the SHACL variant used in this thesis, the corresponding description logic is \mathcal{ALCOIQ} , for which finite satisfiability is known to be decidable [61]. Thus, the translation can be used to decide containment between SHACL shapes.

6.3. Deciding Shape Containment using Standard Entailment

While shape containment can be decided using finite model reasoning, practical usability of the approach depends on whether existing reasoner implementations can be leveraged. Implementations that are readily-available rely on standard entailment, which includes infinitely large models. This thesis therefore also investigates the soundness and completeness of the approach using the standard entailment relation.

For certain description logics, there is a guarantee that, if a model exists, it is a finite one. In these cases, standard entailment and finite model reasoning are the same.

Definition 19 (Finite Model Property). *A description logic has the finite model property if every concept that is satisfiable with respect to a knowledge base has a finite model [19, 84].*

If C is a concept expression that is satisfiable with respect to some knowledge base K that belongs to a description logic having the finite model property, then there must be a finite model of K that shows the satisfiability of C .

As mentioned earlier in Section 6.2, a knowledge base K^S built from a set of shapes S belongs to the description logic \mathcal{ALCOIQ} . The finite model property does not hold for the description logic \mathcal{ALCOIQ} . However, it is possible to restrict SHACL such that the corresponding description logic has the finite model property. In particular, one needs to restrict the use of relations to only allow for forward-facing relations (denoted by p). That is, assume a set of SHACL shapes $S_{\text{non-inv}}$ in which constraints ϕ and queries for target nodes \hat{q} are restricted to the grammar

$$\begin{aligned}\phi_{\text{non-inv}} &::= \top \mid s \mid o \mid \phi \wedge \phi \mid \geq_n p.\phi \\ \hat{q}_{\text{non-inv}} &::= \perp \mid \{\bar{o}\} \mid x_1 \leftarrow x_1 \text{ type concept} \mid x_1 \leftarrow x_1 \text{ property } x_2\end{aligned}$$

As we do not need to cover inverse roles \mathcal{I} anymore, the corresponding description logic of $S_{\text{non-inv}}$ is \mathcal{ALCOQ} . Figure 6.2 shows syntax and semantics of concept expressions in the description logic \mathcal{ALCOQ} . Syntax and semantics of axioms do not change.

For the description logic \mathcal{ALCOQ} , it is known that it has the finite model property. If a concept C is satisfiable with respect to a knowledge base K written in \mathcal{ALCOQ} , then there is a finite model showing the satisfiability of C with respect to K .

Proposition 4. *The description logic \mathcal{ALCOQ} has the finite model property [69].*

Subsequently, for the subset of SHACL shapes $S_{\text{non-inv}}$, the problems of shape containment and concept subsumption in the knowledge base constructed from $S_{\text{non-inv}}$ are equivalent.

Constructor Name	Syntax	Semantics
atomic property	p	$p^I \subseteq \Delta^I \times \Delta^I$
atomic concept	A	$A^I \subseteq \Delta^I$
nominal concept	$\{o\}$	$\{o^I\}$
top	\top	Δ^I
negation	$\neg C$	$\Delta^I \setminus C^I$
conjunction	$C \sqcap D$	$C^I \cap D^I$
qualified number restriction	$\geq n p.C$	$\{o \mid \{o' \mid (o, o') \in p^I \wedge o' \in C^I\} \geq n\}$

Figure 6.2.: Syntax and Semantics of properties p and concept expressions C, D in the description logic \mathcal{ALCOQ} .

Theorem 12. *Let $S_{non-inv}$ be a set of shapes constructed using the constraint grammar $\phi_{non-inv}$. The associated description logic is \mathcal{ALCOQ} . Let $\tau_{shapes}(S_{non-inv}) = K^{S_{non-inv}}$ be the knowledge base constructed from $S_{non-inv}$. Then it holds that*

$$s <:_S s' \text{ with } s, s' \in \text{Names}(S_{non-inv}) \Leftrightarrow K^{S_{non-inv}} \models \tau_{name}(s) \sqsubseteq \tau_{name}(s')$$

Proof. As described in Theorem 11, if there is an RDF data graph and an assignment that acts as a counterexample for s being subsumed by s' , then it can be translated into a finite model that shows that $K^{S_{non-inv}} \not\models \tau_{name}(s) \sqsubseteq \tau_{name}(s')$. On the other hand, there may be a model that acts as a counterexample showing that $K^{S_{non-inv}} \not\models \tau_{name}(s) \sqsubseteq \tau_{name}(s')$. Since \mathcal{ALCOQ} has the finite model property (c. f. Proposition 4), there must be a finite model that can also be used as a counterexample. Therefore, a model exists that can be translated into an RDF data graph and an assignment such that s is not subsumed by s' . \square

For the full definitions of SHACL as used in this thesis however, the corresponding description logic is \mathcal{ALCOIQ} . That is, it is possible to use inverse role expressions in constraints. In particular, they are necessary as Chapter 5 makes use of them in several places, for example when deriving shapes from queries. The finite model property does not hold for \mathcal{ALCOIQ} .

Proposition 5. *The finite model property does not hold for the description logic \mathcal{ALCOIQ} . If a concept C is satisfiable with respect to a knowledge base K written in \mathcal{ALCOIQ} , then it may be that there are only models with an infinitely large universe.*

To highlight Proposition 5, consider the following example adapted from [28]:

$$K_{infinite} = \left\{ \begin{array}{l} \text{Student} \sqsubseteq \exists \text{tutors.Student} \sqcap \leq 1 \text{tutors}^- . \top, \\ \text{BrilliantStudent} \sqsubseteq \text{Student} \sqcap \leq 0 \text{tutors}^- . \top \end{array} \right\}$$

Each `Student` tutors some other `Student` and is tutored by at most one. Furthermore, a `BrilliantStudent` is a `Student` whom no one tutors. The concept `BrilliantStudent` is satisfiable, but not finitely satisfiable. An object that is an instance of `BrilliantStudent` must have a second object that it can tutor. This second object must be an instance of `Student` which means that it must tutor another instance of `Student`. This leads to an infinite sequence of students.

Subsequently, given Proposition 5, it may be that only models with an infinitely large universe exist that show the satisfiability of a concept expression. We can distinguish three different possibilities:

	Finitely Satisfiable	Infinitely Satisfiable	Result
$\tau_{\text{name}}(s) \sqcap \neg\tau_{\text{name}}(s')$	Yes	Yes	$K^S \not\models \tau_{\text{name}}(s) \sqsubseteq \tau_{\text{name}}(s')$
$\tau_{\text{name}}(s) \sqcap \neg\tau_{\text{name}}(s')$	No	Yes	$K^S \not\models \tau_{\text{name}}(s) \sqsubseteq \tau_{\text{name}}(s')$
$\tau_{\text{name}}(s) \sqcap \neg\tau_{\text{name}}(s')$	No	No	$K^S \models \tau_{\text{name}}(s) \sqsubseteq \tau_{\text{name}}(s')$

Two important observations can be made: First, if $K^S \models \tau_{\text{name}}(s) \sqsubseteq \tau_{\text{name}}(s')$, then it means that neither a finitely nor an infinitely large model exists. Second, if $K^S \not\models \tau_{\text{name}}(s) \sqsubseteq \tau_{\text{name}}(s')$, then it may be that there is only an infinitely large model which acts as a counterexample. In this case, shape s would be contained in shape s' for all possible RDF graphs, since the only counterexample found by the corresponding knowledge base has no corresponding RDF graph. Therefore, for the DL \mathcal{ALCOIQ} , deciding shape containment is sound but incomplete.

Theorem 13. *Let S be a set of shapes constructed by using the constraint grammar ϕ and the target node grammar \hat{q} . Deciding shape containment is sound but incomplete, as it holds that:*

$$s <:_S s' \text{ with } s, s' \in \text{Names}(S) \Leftarrow \tau_{\text{shapes}}(S) \models \tau_{\text{name}}(s) \sqsubseteq \tau_{\text{name}}(s')$$

Proof. For the complete SHACL definitions, the corresponding description logic is \mathcal{ALCOIQ} for which the finite model property does not hold. If $K^S \models \tau_{\text{name}}(s) \sqsubseteq \tau_{\text{name}}(s')$, then neither finitely nor infinitely large models exist. The shape s must therefore be contained in the shape s' as there is no RDF graph and assignment that acts as a counterexample. \square

Subsequently, deciding shape containment in a set of shapes S via a translation to a description logic knowledge base is sound—that is, if a knowledge base infers that $K^S \models \tau_{\text{name}}(s) \sqsubseteq \tau_{\text{name}}(s')$, then it must be that shape s is contained in s' . However, it is also incomplete as $K^S \not\models \tau_{\text{name}}(s) \sqsubseteq \tau_{\text{name}}(s')$ does not mean that it is possible to find a counterexample for s being contained in s' .

6.4. Effects on Algorithmic Type Checking for λ_{SHACL}

For SHACL constraints that support inverse roles, a translation of the shape containment problem into a description logics subsumption problem provides a sound but incomplete approach to deciding

shape subsumption. In case of λ_{SHACL} however, it provides a considerable improvement over the `SUBTYPE` predicate as shown in Algorithm 12. While λ_{SHACL} is dependent on inverse role expressions for typing projections and queries, the new approach to shape containment allows for using the complete constraint grammar. The `SUBTYPING` predicate therefore simply translates the set of shapes into a description logics knowledge base using a function `TRANSLATE` and then uses established DL reasoners to decide containment (see Algorithm 14).

Algorithm 14 Improved definition of `SUBTYPE` for λ_{SHACL} .

```

function SUBTYPE( $S, T, T'$ )
  match ( $T, T'$ ) with
    case ... then ...      ▷ existing cases
    case ( $s, s'$ ) when  $\tau_{\text{shapes}}(S) \models \tau_{\text{name}}(s) \sqsubseteq \tau_{\text{name}}(s')$  then true
    case _ then FALSE

```

Algorithm 15 Improved definition of `LUB` for λ_{SHACL} .

```

global variables
   $S$ , a set of shapes
function LUB( $T, T'$ )
  match ( $T, T'$ ) with
    case ... then ...      ▷ existing cases
    case ( $s, s'$ ) when ( $s, \phi, \hat{q}$ )  $\in S$  and ( $s', \phi', \hat{q}'$ )  $\in S$  and  $s_{\text{lub}}$  is GENSHAPENAME() then
       $S \leftarrow S \cup \{(s_{\text{LUB}}, \phi \vee \phi', \perp)\}$ 
       $s_{\text{LUB}}$ 
    case _ then TOP

```

Using the complete constraint grammar also allows us to update the least upper bound `LUB` as we can now express disjunction between two constraints. The least-upper bound `LUB` simply creates a new shape that combines the constraints of the two input shapes using disjunction. The greatest-lower bound `GLB` does not change (see Algorithm 15).

6.5. Summary and Discussion

In this chapter, we have considered shape containment between SHACL shapes via a translation to description logics. Results show that for the subset of SHACL that does not support inverse roles or even more advanced property path expressions, the reduction to description logics provides a sound and complete way of deciding shape containment. In case of more expressive constraints that include

Chapter 6. Shape Containment

inverse role expressions or target node queries that query for objects of properties, the reduction to description logics remains sound but is incomplete. This in turn allows for an improved implementation of the type system described in Chapter 5.

Related Work

This work is related to several areas of research. First, we are related to *RDF Schema Languages* in general. Second, we are concerned with providing types for queries that require a subtype relation in the type system. As the subtype relation indicates a subset relation between sets of values, this brings us to the field of *containment problems*. Lastly, our work integrates new types into programming languages. As such, we are related to the field of *Language Integration* and the various approaches used there.

7.1. RDF Schema Languages

Several schema languages exist for RDF. First, there is the W3C recommendation RDF Schema (RDFS) [6]. However, RDFS is not a schema language in the validation sense (c. f., JSON Schema [5] or XML Schema [42]) but rather an ontology language that allows inferring implicit facts using a fixed-point semantics that applies the rules as defined in the ontology until a fixpoint has been reached. Similarly, the Web Ontology Language (OWL) [47] is an ontology language that is rooted in description logics [18] featuring a possible world semantics. It allows for modeling incomplete knowledge by interpreting missing relations and nodes as implicit. As shown in Chapter 4, it can be used for type-checking. The type system ensures that the data is used according to the conventions defined in the ontology—for example, that only relations are accessed for which it is known that a node has them. However, a developer must still consider implicit knowledge when writing a program. Even if a node is an instance of a concept $\exists \text{ studiesAt}.\top$, it may be that accessing the --studiesAt-- relation for the node yields an empty list.

In terms of dedicated constraint mechanisms, several approaches exist. [41] introduces constraints into RDF that are evaluated via a translation to SPARQL, whereas [16] evaluates constraints using chasing algorithms as used in database theory. For description logics, several proposals have been made. Constraints can be introduced as new forms of special constraint axioms [90, 71]. Another approach is to introduce an epistemic operator into the construction of axioms [38]. Lastly, of certain predicates can be marked as complete [79]. To the best of our knowledge, Stardog [11] is the only triplestore that allows for using both, standard DL axioms and constraints axioms. Constraint axioms

are evaluated by a translation to SPARQL (similar to [90]) in this case.

In terms of dedicated constraint languages, SPIN [10] allows for expressing rules and constraints as SPARQL queries. It is officially superseded by SHACL since 2017 [3]. Shex [9] is a constraint language inspired by XML schema languages. Contrary to the SHACL semantics [35] used in this thesis, ShEx does not rely on a possible world semantics. While ShEx supports everything required for a type system as defined in Chapter 5, we chose SHACL due to it being a W3C recommendation.

7.2. Containment Problems

Containment problems are well known in the context of types, such as highlighted by typing for XML and its schema languages [56, 62, 30]. In the context of RDF, containment has been investigated for ShEx [87] schemata, which serves the same purpose as SHACL but is inspired by XML schema languages. The containment problem in ShEx is, e. g., comparable to the containment of regular expressions.

SHACL without recursion can be evaluated through queries [34]. Subsequently, shapes can be represented as queries. In this case, containment of shapes is equivalent to query containment. Again, the containment problem for queries is well known, for highly expressive query languages [26] as well as containment for queries with constraints imposed by some form of schema [29, 55]. However, the SHACL subset used in this thesis is too expressive to allow for deciding containment of SHACL shapes through query containment.

Chapter 6 instead used a translation of the SHACL shape containment problem into a description logic subsumption problem. Subsumption problems in description logics are a fundamental operation. Depending on the expressiveness of the description logic, various algorithms for deciding subsumption exist (e. g. [25, 54]) as well as various optimization techniques [52]. Most approaches are based on semantic tableaux [19].

7.3. Language Integration

The integration of data models into programming languages is a well-known problem. In general, we consider four different ways of integrating a data model into a programming language: By using generic representations, by mappings into the target language, through a preprocessing step before compilation, or through language extensions and custom languages.

Generic Representations Generic representations offer easy integration into programming languages and have the advantage that they can represent anything the data can model. They rely on types on a meta-level such as *Axiom* (e. g. [51]), *Node* [32], or *Statement* [2]. In that, they are comparable

to generic representations such as DOM [1] for XML [92]. However, types on a meta-level do not allow a static type-checker to verify a program with respect to the data. This leaves correctness entirely in the hands of the programmer.

Mappings Mapping approaches use schematic information of the data model to create types in the target language. Type checking therefore check the valid use of the derived types in programs. This approach has been successfully used for SQL [75], XML [92, 63, 17]. [63, 89] describe more general approaches. Naturally, mappings have been studied in a semantic data context, too. The focus is on transforming conceptual statements of schema languages into types of the programming language. Frameworks include ActiveRDF [76], Owl2Java [58], Jastor [4] RDFReactor [8], Àgogo [78] and LITEQ [66]. An important point is that all these approaches work on ontologies. So far, no approach exists that makes use of SHACL. The do, however, provide some level of type-checking that can help in avoiding errors. On the other hand, mapping approaches for ontologies are also limited. OWL ontologies may mix structural and nominal typing whereas types in programming languages typically use only one of the two approaches. Ontologies often provide only extremely general information on domain and range of relation occur frequently. For example, the relation `-foaf:made-` is defined such that the domain is an `(Agent)` and the range a `(Thing)`. Frameworks often resolve such situations by assigning these relations to every type they create whereas they usually assign the most general type as a result type. They then leave it to the developer to cast values to their correct type. This is an error-prone approach. Lastly, all mapping frameworks have problems with the large number of potential types in semantic data sources.

Precompilation A separate precompilation step, where the source code is statically analyzed and then transformed, is another way to solve the problem of integrating data models into programming languages. Especially queries embedded in programming languages can be verified in this manner. While this approach can verify a program with respect to the data, interaction between the language and the newly integrated data model is problematic. Furthermore, complexity arises from the fact that the transformation needs to preserve all constructs supported by the host language. The approach has been applied to, for example, SQL queries [93]. Such an approach has also been applied for SPARQL in a limited manner [46]—however, only for queries that can be typed with primitive types such as integer. Queries returning graph nodes are typed generically as *Resource*.

Language extensions and custom languages The most powerful approaches create new languages or extend existing ones to accommodate the specific requirements of the data model. While they are complex, they allow for total control over typing, subtyping and interaction between programming language and data model. Examples for such extensions are concerned with relationships and XML data [23] and easy data access to relational and XML data [22]. Another example concerns

the programming language support for the XML data model specifically in terms of regular expression types, as in the languages CDuce [20] and XDuce [56]. While RDF graphs could be seen as simple XML documents, the XML-focused approaches do not address challenges that arise from the possible world semantics used by the schema language as used for RDF data. Similarly, polymorphic record types in object-oriented database systems [74] are oriented towards structural typing. For data in the Semantic Web, a mixture between nominal and structural typing is needed. Refinement types, e. g., as provided by F* [88] are somewhat closer to the types as defined in this thesis. They allow for capturing pre- and postconditions of functions at the type level and to verify correctness statically. By contrast, DL expressions are logic formulae over nominal and structural type properties. They define new types which are subject to DL-based reasoning for type checking leveraging their possible worlds semantics. Similarly, SHACL shapes cannot be fully encoded in F* due to the possible world semantics. Another related approach is the idea of functional logic programming [48]. However, this thesis emphasizes type-checking on data axiomatized in logic over the integration of the logic programming paradigm into a language.

The typecase construct of λ_{DL} as defined in Chapter 4 is inspired by other forms of typecase such as those in the context of dynamic typing [13], intensional polymorphism [36], and generic functional programming [64]. None of these forms are concerned with RDF data or RDF schema languages.

Language extensions and custom approaches have also been implemented for semantic data. Examples include rule-based programming [57] as well as a transformation and validation language [82]. However, both are untyped. A typed approach to linked data—a specific kind of semantic data—is provided by [50, 33]. Similarly, Zhi# [77], an extension of the C# language provides an integration for OWL ontologies, albeit it only considers explicitly given statements. To the best of our knowledge, only implementations of λ_{DL} [85, 49], as described in Chapter 4, consider implicit statements during the type checking process.

Conclusion

Summary In this thesis, we developed two type-safe languages, λ_{DL} and λ_{SHACL} , for working with RDF data graphs as used in the Semantic Web. We leveraged two different schema languages for this: The Ontology Web Language (OWL), or rather its theoretic foundations in the form of description logics, and the Shape Constraint Language (SHACL). Both languages feature new forms of types and typed data access.

In case of description logics, we used concept expressions as new forms of types. As they are syntactic symbols that semantically represent sets of objects (or graph nodes), this was a natural choice. While type-safety was achieved, several noteworthy effects come into play due to the open-world assumption employed by description logics. First, the typecase construct has the interesting requirement of needing a default-case. This is because we require certain knowledge. For a given node, it is certain that it is an instance of `Student \sqcup \neg Student`. However, when we first ask whether the node is an instance of `Student` and then ask whether the node is an instance of `\neg Student`, it may be that the answer is `false` in both cases. This is because there may be some models that satisfy the data and schematic restrictions, but which do not agree whether the graph node is an instance of `Student`. This is unusual behavior compared to other existing typecase constructs. Second, given an object `bob` that is an instance of the concept expression `\exists hasName. \top` , it may still be that the programming language term `o.hasName` yields an empty list. This is because description logics allow for modeling unknown knowledge. It may be that it is known that a place of study must exist, but is currently not contained in the data. This behavior is contrary to what a developer typically expects.

This prompted the investigation of a second type system based on a fixed domain. For this, we leveraged SHACL. While shape names are used as type names, they semantically represent the set of nodes conforming to the shape. Assuming a node `(bob)` conforms to a shape that guarantees the presence of a `-hasName \rightarrow` relation, the term `o.hasName` where `o` represents `(bob)` will yield results. However, SHACL so far does not support implicit knowledge that can be derived using ontologies. Furthermore, subtyping between shapes requires deciding shape containment—which is an open issue. Subsequently, subtyping in λ_{SHACL} is limited.

Lastly, we investigated a possible translation of shape containment to concept subsumption in description logics. In general, shape containment can be decided via counterexamples. A shape `s` must

be contained in a shape s' if it is not possible to find a counterexample. Likewise, description logics decide concept subsumption through counterexamples. Given a set of shapes, it is possible to define an equivalent knowledge base such that atomic concepts represent shape names. In this case, a finite model of that knowledge base is equivalent to an RDF graph and a faithful assignment in SHACL. If a finite model is a counterexample to concept subsumption, then it also acts as a counterexample for shape containment. This allows implementations of λ_{SHACL} to leverage description logic reasoners for subtyping.

Future Work The presented approach can be extended in several directions.

Combining λ_{DL} and λ_{SHACL} In practice, a developer probably wants the best of both worlds. The ability to derive implicit facts from given knowledge using OWL ontologies is just as desirable as having concrete guarantees about the data as provided by SHACL. Future work should, therefore investigate a type system that combines both—OWL and SHACL—and allows for using either DL concept expressions or SHACL shapes as types, depending on the situation. Most importantly, this requires the ability to use subtyping between shape names and concept expressions.

Enhanced Querying While the query language chosen in this thesis is the most commonly used SPARQL fragment [80], it only supports a small subset of SPARQL features. Advanced features such as disjunction, filter expressions or more powerful property path expressions should be considered. However, this must be done carefully to avoid undecidability in the underlying description logic.

Polymorphism Polymorphism is a standard feature in modern programming languages. In particular ad-hoc polymorphism and function overloading, meaning that a function defined over several types can act differently depending on the type, is challenging in case of working with RDF data graphs. Any graph node belongs to many different types that may not be in a hierarchy. Subsequently, future work should investigate how ad-hoc polymorphism and function overloading can be defined—in particular, how it can be decided which function should be applied and how conflicting cases that require the decision of a developer can be identified.

Modification of data Lastly, neither λ_{DL} nor λ_{SHACL} have any way of modifying data. However, applications often do not only consume data but also modify existing data or insert new data points. The type system should also support these kinds of operations and give guarantees about the changes made by a program—e. g., that inserting new data will not violate existing SHACL constraints.

Implementations of λ_{DL} and λ_{SHACL} including tooling While we did integrate λ_{DL} into Scala (c. f. [85]) and do plan to do so for λ_{SHACL} as well, application development is typically more than just a programming language. Developers rely on powerful tools in IDEs such as autocompletion. It remains

future work to investigate how λ_{DL} and λ_{SHACL} can be integrated into programming languages such that IDEs and other tools do not break.

A user study of λ_{DL} and λ_{SHACL} While both languages λ_{DL} and λ_{SHACL} are sound from a theoretic point of view, their impact on practical application development remains to be seen. While static type systems appear to improve productivity [39], a user study is required to verify that the additional checks of the type system outweigh the restrictions on the expressiveness in case of programming with RDF data graphs.

Bibliography

- [1] Document Object Model (DOM), <https://www.w3.org/DOM/>
- [2] Eclipse rdf4j, <http://rdf4j.org/>
- [3] From SPIN To SHACL, <https://spinrdf.org/spin-shacl.html>
- [4] Jastor: Typesafe, Ontology Driven RDF Access from Java, <http://jastor.sourceforge.net/>
- [5] json-schema.org: The home of JSON Schema, <http://json-schema.org/>
- [6] RDF Schema 1.1, <https://www.w3.org/TR/rdf-schema/>
- [7] RDFox, <https://www.cs.ox.ac.uk/isg/tools/RDFox/>
- [8] Semweb4j: RDF2Go and RDFReactor, <https://www.aifb.kit.edu/web/Semweb4j>
- [9] ShEx: Shape Expressions, <https://shex.io/>
- [10] SPIN: SPARQL Inferencing Notation, <https://spinrdf.org/>
- [11] Stardog: The Enterprise Knowledge Graph Platform, <https://www.stardog.com/>
- [12] Abadi, M., Cardelli, L.: An Imperative Object Calculus. In: Proceedings of TAPSOFT: Theory and Practice of Software Development. pp. 471–485. LNCS, Springer (1995). doi: 10.1007/3-540-59293-8_214
- [13] Abadi, M., Cardelli, L., Pierce, B.C., Rémy, D.: Dynamic Typing in Polymorphic Languages. Journal Functional Programming 5(1), 111–130 (1995). doi: 10.1017/S09567968000126X
- [14] Abbas, A., Genevès, P., Roisin, C., Layaïda, N.: Optimising SPARQL Query Evaluation in the Presence of ShEx Constraints. In: BDA 2017 - 33ème conférence sur la “ Gestion de Données - Principes, Technologies et Applications ”. pp. 1–12 (Nov 2017)
- [15] Abbas, A., Genevès, P., Roisin, C., Layaïda, N.: SPARQL Query Containment with ShEx Constraints. In: Proceedings of the Advances in Databases and Information Systems (ADBIS 2017). LNCS, vol. 10509, pp. 343–356. Springer (2017). doi: 10.1007/978-3-319-66917-5_23

Bibliography

- [16] Akhtar, W., Cortés-Calabuig, A., Paredaens, J.: Constraints in RDF. In: *Semantics in Data and Knowledge Bases - 4th International Workshops (SDKB 2010)*. LNCS, vol. 6834, pp. 23–39. Springer (2010). doi: 10.1007/978-3-642-23441-5_2, https://doi.org/10.1007/978-3-642-23441-5_2
- [17] Alagic, S., Bernstein, P.A.: Mapping XSD to OO Schemas. In: *Object Databases, Second International Conference (ICOODB 2009)*. LNCS, vol. 5936, pp. 149–166. Springer (2009). doi: 10.1007/978-3-642-14681-7_9
- [18] Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F. (eds.): *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press (2003)
- [19] Baader, F., Horrocks, I., Lutz, C., Sattler, U.: *An Introduction to Description Logic*. Cambridge University Press, 1st edn. (2017)
- [20] Benzaken, V., Castagna, G., Frisch, A.: CDuce: An XML-centric General-purpose Language. In: *Proceedings of the 8th International Conference on Functional Programming (ICFP 2003)*. pp. 51–63. ACM (2003). doi: 10.1145/944705.944711
- [21] Berners-Lee, T., Hendler, J., Lassila, O.: The Semantic Web. *Scientific American* 284(5), 34–43 (May 2001)
- [22] Bierman, G.M., Meijer, E., Schulte, W.: The Essence of Data Access in Comega. In: *Proceedings of the 19th Conference on Object-Oriented Programming (ECOOP 2005)*. LNCS, vol. 3586, pp. 287–311. Springer (2005). doi: 10.1007/11531142_13
- [23] Bierman, G.M., Wren, A.S.: First-Class Relationships in an Object-Oriented Language. In: *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP 2005)*. LNCS, vol. 3586, pp. 262–286. Springer (2005). doi: 10.1007/11531142_12
- [24] Bischof, S., Krötzsch, M., Polleres, A., Rudolph, S.: Schema-Agnostic Query Rewriting in SPARQL 1.1. In: *Proceedings of the 13th International Semantic Web Conference (ISWC)*. pp. 584–600. LNCS, Springer (2014). doi: 10.1007/978-3-319-11964-9_37
- [25] Borgida, A., Patel-Schneider, P.F.: A Semantics and Complete Algorithm for Subsumption in the CLASSIC Description Logic. *Journal of Artificial Intelligence Research* 1, 277–308 (1994). doi: 10.1613/jair.56
- [26] Bourhis, P., Krötzsch, M., Rudolph, S.: Reasonable Highly Expressive Query Languages. In: *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI 2015)*. pp. 2826–2832. AAAI Press (2015)

- [27] Bourhis, P., Reutter, J.L., Suárez, F., Vrgoč, D.: JSON: Data Model, Query Languages and Schema Specification. In: Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems. pp. 123–135. PODS '17, ACM (2017). doi: 10.1145/3034786.3056120
- [28] Calvanese, D.: Finite Model Reasoning in Description Logics. In: Proceedings of the 5th International Conference on Principles of Knowledge Representation and Reasoning (KR 1996). pp. 292–303. Morgan Kaufmann (1996)
- [29] Calvanese, D., De Giacomo, G., Lenzerini, M.: On the Decidability of Query Containment under Constraints. In: Proceedings of the 17th Symposium on Principles of Database Systems (PODS 1998). pp. 149–158. ACM Press (1998). doi: 10.1145/275487.275504
- [30] Calvanese, D., De Giacomo, G., Lenzerini, M.: Representing and Reasoning on XML Documents: A Description Logic Approach. *Journal of Logic and Computation* **9**(3), 295–318 (1999). doi: 10.1093/logcom/9.3.295
- [31] Cardelli, L., Wegner, P.: On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys* **17**(4), 471–523 (Dec 1985)
- [32] Carroll, J.J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., Wilkinson, K.: Jena: implementing the semantic web recommendations. In: Proceedings of the 13th Conference on World Wide Web - Alternate Track Papers & Posters (WWW 2004). pp. 74–83. ACM (2004). doi: 10.1145/1013367.1013381
- [33] Ciobanu, G., Horne, R., Sassone, V.: Minimal type inference for Linked Data consumers. *Journal of Logical and Algebraic Methods in Programming* **84**(4), 485–504 (2015). doi: 10.1016/j.jlamp.2014.12.005
- [34] Corman, J., Florenzano, F., Reutter, J.L., Savkovic, O.: Validating Shacl Constraints over a Sparql Endpoint. In: Proceedings of the 18th International Semantic Web Conference (ISWC 2019). LNCS, vol. 11778, pp. 145–163. Springer (2019). doi: 10.1007/978-3-030-30793-6_9
- [35] Corman, J., Reutter, J.L., Savkovic, O.: Semantics and Validation of Recursive SHACL. In: Proceedings of the 17th International Semantic Web Conference (ISWC). LNCS, vol. 11136, pp. 318–336. Springer (2018). doi: 10.1007/978-3-030-00671-6_19
- [36] Crary, K., Weirich, S., Morrisett, J.G.: Intensional polymorphism in type-erasure semantics. *Journal of Functional Programming* **12**(6), 567–600 (2002). doi: 10.1017/S0956796801004282
- [37] Cyganiak, R., Wood, D., Lanthaler, M.: RDF 1.1 Concepts and Abstract Syntax. W3C Recommendation (2014), <https://www.w3.org/TR/rdf11-concepts/>

Bibliography

- [38] Donini, F.M., Nardi, D., Rosati, R.: Description Logics of Minimal Knowledge and Negation As Failure. *ACM Transactions on Computational Logic (TOCL)* 3(2), 177–225 (Apr 2002). doi: 10.1145/505372.505373
- [39] Endrikat, S., Hanenberg, S., Robbes, R., Stefik, A.: How do API documentation and static typing affect API usability? In: *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. pp. 632–642. ACM (2014). doi: 10.1145/2568225.2568299
- [40] Fernandez, M.F., Suciu, D.: Optimizing Regular Path Expressions Using Graph Schemas. In: *Proceedings of the 14th International Conference on Data Engineering*. pp. 14–23. IEEE Computer Society (1998). doi: 10.1109/ICDE.1998.655753
- [41] Fischer, P.M., Lausen, G., Schätzle, A., Schmidt, M.: RDF Constraint Checking. In: *Proceedings of the Workshops of the EDBT/ICDT 2015 Joint Conference (EDBT/ICDT)*. CEUR Workshop Proceedings, vol. 1330, pp. 205–212. CEUR-WS.org (2015), <http://ceur-ws.org/Vol-1330/paper-33.pdf>
- [42] Gao, S.S., Sperberg-McQueen, C.M., Thompson, H.S.: W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures. W3C Recommendation (2012), <https://www.w3.org/TR/xmlschema11-1/>
- [43] Gerakios, P., Fourtounis, G., Smaragdakis, Y.: Foo: a minimal modern OO calculus. In: *Proceedings of the 17th Workshop on Formal Techniques for Java-like Programs (FTfJP)*. pp. 3:1–3:4. ACM (2015). doi: 10.1145/2786536.2786540
- [44] Glew, N.: Type Dispatch for Named Hierarchical Types. In: *Proceedings of the International Conference on Functional Programming (ICFP)*. pp. 172–182. ACM (1999). doi: 10.1145/317636.317797
- [45] Glimm, B., Ogbuji, C.: SPARQL 1.1 Entailment Regimes. W3C Recommendation (2013), <https://www.w3.org/TR/sparql11-entailment/>
- [46] Grope, S., Neumann, J., Linnemann, V.: SWOBE - Embedding the Semantic Web Languages RDF, SPARQL and SPARUL into Java for Guaranteeing Type Safety, for Checking the Satisfiability of Queries and for the Determination of Query Result Types. In: *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC 2009)*. pp. 1239–1246. ACM (2009). doi: 10.1145/1529282.1529561
- [47] Group, W.O.W.: OWL 2 Web Ontology Language: Document Overview. W3C Recommendation (2012), <https://www.w3.org/TR/owl2-overview/>
- [48] Hanus, M.: The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming* 19/20, 583–628 (1994). doi: 10.1016/0743-1066(94)90034-5, [https://doi.org/10.1016/0743-1066\(94\)90034-5](https://doi.org/10.1016/0743-1066(94)90034-5)

- [49] Hartenfels, C., Leinberger, M., Lämmel, R., Staab, S.: Type-Safe Programming with OWL in Semantics4J. In: Proceedings of the ISWC 2017 Posters & Demonstrations and Industry Tracks, 16th International Semantic Web Conference (ISWC 2017). CEUR Workshop Proceedings, vol. 1963. CEUR-WS.org (2017), <http://ceur-ws.org/Vol-1963/paper549.pdf>
- [50] Horne, R., Sassone, V.: A verified algebra for read-write Linked Data. *Science of Computer Programming* **89**, 2–22 (2014). doi: 10.1016/j.scico.2013.07.005
- [51] Horridge, M., Bechhofer, S.: The OWL API: A Java API for OWL ontologies. *Semantic Web* **2**(1), 11–21 (2011)
- [52] Horrocks, I.: Implementation and Optimization Techniques. In: *The Description Logic Handbook: Theory, Implementation, and Applications*, pp. 306–346. Cambridge University Press (2003)
- [53] Horrocks, I.: Ontologies and the Semantic Web. *Communications of the ACM* **51**(12), 58–67 (Dec 2008). doi: 10.1145/1409360.1409377
- [54] Horrocks, I., Kutz, O., Sattler, U.: The Even More Irresistible SROIQ. In: Proceedings of the 10th International Conference on Principles of Knowledge Representation and Reasoning (KR). pp. 57–67. AAAI Press (2006)
- [55] Horrocks, I., Sattler, U., Tessaris, S., Tobies, S.: How to Decide Query Containment under Constraints Using a Description Logic. In: *Logic for Programming and Automated Reasoning*. pp. 326–343. Springer (2000)
- [56] Hosoya, H., Pierce, B.C.: XDuce: A Statically Typed XML Processing Language. *ACM Transactions on Internet Technology (TOIT)* **3**(2), 117–148 (May 2003). doi: 10.1145/767193.767195
- [57] Käfer, T., Harth, A.: Rule-based Programming of User Agents for Linked Data. In: *Workshop on Linked Data on the Web*. CEUR Workshop Proceedings, vol. 2073. CEUR-WS.org (2018), <http://ceur-ws.org/Vol-2073/article-05.pdf>
- [58] Kalyanpur, A., Pastor, D.J., Battle, S., Padget, J.A.: Automatic Mapping of OWL Ontologies into Java. In: *Proceedings of the 16th International Conference on Software Engineering & Knowledge Engineering (SEKE'2004)*. pp. 98–103 (2004)
- [59] Knublauch, H., et al.: Shapes Constraint Language (SHACL). W3C Recommendation (2017), <https://www.w3.org/TR/shacl/>
- [60] Kollia, I., Glimm, B., Horrocks, I.: Query Answering over SROIQ Knowledge Bases with SPARQL. In: *Proceedings of the 24th International Workshop on Description Logics (DL 2011)*. CEUR Workshop Proceedings, vol. 745. CEUR-WS.org (2011)

Bibliography

- [61] Kotek, T., Simkus, M., Veith, H., Zuleger, F.: Extending ALCQIO with Trees. In: Proceedings of the 30th Symposium on Logic in Computer Science (LICS). pp. 511–522. IEEE Computer Society (2015). doi: 10.1109/LICS.2015.54
- [62] Kuper, G.M., Siméon, J.: Subsumption for XML types. In: Proceedings of the 8th International Conference on Database Theory (ICDT 2001). LNCS, vol. 1973, pp. 331–345. Springer (2001). doi: 10.1007/3-540-44503-X_21
- [63] Lämmel, R., Meijer, E.: Revealing the X/O Impedance Mismatch - (Changing Lead into Gold). In: Datatype-Generic Programming - International Spring School (SSDGP 2006). LNCS, vol. 4719, pp. 285–367. Springer (2006). doi: 10.1007/978-3-540-76786-2_6
- [64] Lämmel, R., Peyton Jones, S.L.: Scrap your boilerplate: a practical design pattern for generic programming. In: Proceedings the International Workshop on Types in Languages Design and Implementation (TLDI 2003). pp. 26–37. ACM (2003). doi: 10.1145/604174.604179
- [65] Leinberger, M., Lämmel, R., Staab, S.: The Essence of Functional Programming on Semantic Data. In: Proceedings of the 26th European Symposium on Programming (ESOP 2017). LNCS, vol. 10201, pp. 750–776. Springer (2017). doi: 10.1007/978-3-662-54434-1_28
- [66] Leinberger, M., Scheglmann, S., Lämmel, R., Staab, S., Thimm, M., Viegas, E.: Semantic Web Application Development with LITEQ. In: Proceedings of the 13th International Semantic Web Conference (ISWC 2014). LNCS, vol. 8797, pp. 212–227. Springer (2014). doi: 10.1007/978-3-319-11915-1_14
- [67] Leinberger, M., Seifer, P., Rienstra, T., Lämmel, R., Staab, S.: Deciding SHACL Shape Containment Through Description Logics Reasoning. In: Proceedings of the 19th International Semantic Web Conference (ISWC 2020). LNCS, vol. 12506, pp. 366–383. Springer (2020). doi: 10.1007/978-3-030-62419-4_21
- [68] Leinberger, M., Seifer, P., Schon, C., Lämmel, R., Staab, S.: Type Checking Program Code Using SHACL. In: Proceedings of the 18th International Semantic Web Conference (ISWC 2019). LNCS, vol. 11778, pp. 399–417. Springer (2019). doi: 10.1007/978-3-030-30793-6_23
- [69] Lutz, C., Areces, C., Horrocks, I., Sattler, U.: Keys, nominals, and concrete domains. *Journal of Artificial Intelligence Research* 23, 667–726 (2004). doi: 10.1613/jair.1542
- [70] Milner, R.: A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences* 17(3), 348–375 (1978). doi: 10.1016/0022-0000(78)90014-4
- [71] Motik, B., Horrocks, I., Sattler, U.: Adding Integrity Constraints to OWL. In: Proceedings 2007 Workshop on OWL (OWLED 2007). CEUR Workshop Proceedings, vol. 258. CEUR-WS.org (2007)

- [72] Motik, B., Shearer, R., Horrocks, I.: Hypertableau Reasoning for Description Logics. *Journal of Artificial Intelligence Research* **36**, 165–228 (2009)
- [73] Norvig, P.: The Semantic Web and the Semantics of the Web: Where Does Meaning Come From? In: *Proceedings of the 25th International Conference on World Wide Web (WWW 2016)*. p. 1. ACM (2016). doi: 10.1145/2872427.2874818
- [74] Ogori, A., Buneman, P., Breazu-Tannen, V.: Database Programming in Machiavelli—a Polymorphic Language with Static Type Inference. In: *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data (SIGMOD 1989)*. pp. 46–57. ACM (1989). doi: 10.1145/67544.66931
- [75] O’Neil, E.J.: Object/relational mapping 2008: hibernate and the entity data model (edm). In: *Proceedings of the International Conference on Management of Data (SIGMOD 2008)*. pp. 1351–1356. ACM (2008). doi: 10.1145/1376616.1376773
- [76] Oren, E., Delbru, R., Gerke, S., Haller, A., Decker, S.: ActiveRDF: object-oriented semantic web programming. In: *Proceedings of the 16th International Conference on World Wide Web (WWW 2007)*. pp. 817–824. ACM (2007). doi: 10.1145/1242572.1242682
- [77] Paar, A., Vrandečić, D.: Zhi# - OWL Aware Compilation. In: *Proceedings of the 8th Extended Semantic Web Conference (ESWC 2011)*. LNCS, vol. 6644, pp. 315–329. Springer (2011). doi: 10.1007/978-3-642-21064-8_22
- [78] Parreiras, F.S., Saathoff, C., Walter, T., Franz, T., Staab, S.: à gogo: Automatic Generation of Ontology APIs. In: *Proceedings of the 3rd IEEE International Conference on Semantic Computing (ICSC 2009)*. pp. 342–348. IEEE Computer Society (2009). doi: 10.1109/ICSC.2009.90
- [79] Patel-Schneider, P.F., Franconi, E.: Ontology Constraints in Incomplete and Complete Data. In: *Proceedings 11th International Semantic Web Conference (ISWC 2012)*. LNCS, vol. 7649, pp. 444–459. Springer (2012). doi: 10.1007/978-3-642-35176-1_28
- [80] Picalausa, F., Luo, Y., Fletcher, G.H.L., Hidders, J., Vansummeren, S.: A Structural Approach to Indexing Triples. In: *Proceedings of the 9th International Conference on The Semantic Web (ESWC)*. pp. 406–421. LNCS, Springer (2012). doi: 10.1007/978-3-642-30284-8_34
- [81] Pierce, B.C.: *Types and Programming Languages*. The MIT Press, 1st edn. (2002)
- [82] Prud’hommeaux, E., Gayo, J.E.L., Solbrig, H.R.: Shape expressions: an RDF validation and transformation language. In: *Proceedings of the 10th International Conference on Semantic Systems (SEMANTICS 2014)*. pp. 32–40. ACM (2014). doi: 10.1145/2660517.2660523

Bibliography

- [83] Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF. W3C Recommendation (2013), <https://www.w3.org/TR/rdf-sparql-query/>
- [84] Rudolph, S.: Foundations of Description Logics, pp. 76–136. Springer (2011). doi: 10.1007/978-3-642-23032-5_2
- [85] Seifer, P., Leinberger, M., Lämmel, R., Staab, S.: Semantic Query Integration With Reason. *Programming Journal* 3(3), 13 (2019). doi: 10.22152/programming-journal.org/2019/3/13
- [86] Staab, S., Studer, R.: Handbook on Ontologies. Springer, 2nd edn. (2009)
- [87] Staworko, S., Wiczorek, P.: Containment of Shape Expression Schemas for RDF. In: Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS 2019). pp. 303–319. ACM (2019). doi: 10.1145/3294052.3319687
- [88] Swamy, N., Chen, J., Fournet, C., Strub, P., Bhargavan, K., Yang, J.: Secure distributed programming with value-dependent types. In: Proceeding of the 16th International Conference on Functional Programming (ICFP 2011). pp. 266–278. ACM (2011). doi: 10.1145/2034773.2034811
- [89] Syme, D., Battocchi, K., Takeda, K., Malayeri, D., Petricek, T.: Themes in information-rich functional programming for internet-scale data sources. In: Proceedings of the Workshop on Data Driven Functional Programming (DDFP 2013). pp. 1–4. ACM (2013). doi: 10.1145/2429376.2429378
- [90] Tao, J., Sirin, E., Bao, J., McGuinness, D.L.: Integrity Constraints in OWL. In: Proceedings 24th Conference on Artificial Intelligence (AAAI 2010). AAAI Press (2010)
- [91] Vrandečić, D., Krötzsch, M.: Wikidata: a free collaborative knowledge base. *Communications of the ACM* 57(10), 78–85 (2014). doi: 10.1145/2629489
- [92] Wallace, M., Runciman, C.: Haskell and XML: Generic Combinators or Type-Based Translation? In: Proceedings of International Conference on Functional Programming (ICFP). pp. 148–159. ACM (1999)
- [93] Wassermann, G., Gould, C., Su, Z., Devanbu, P.T.: Static checking of dynamically generated queries in database applications. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 16(4), 14 (2007). doi: 10.1145/1276933.1276935

Soundness of λ_{Full}

The language λ_{Full} as introduced in Chapter 3 is sound. That is, a well-typed program does not get stuck unless it reaches a point where it tries to compute `head nil[T]` or `tail nil[T]` for any type T . For this, we show progress of preservation. The proof is taken from *Types and Programming Languages* [81] with slight modifications. We start by observing the well-typed values that are contained in the language:

Lemma 4. *Let v be a well-typed value. Then one of the following must be true:*

1. If v is a value of type `Bool`, then either $v = \text{true}$ or $v = \text{false}$.
2. If v is a value of type `Nat`, then v is a numerical value `nv` according to the grammar defined in Figure 2.1.
3. If v is a value of type $\{l_i : T_i^{i \in 1 \dots n}\}$ then v must be a record of the form $\{l_j : t_j^{j \in 1 \dots m}\}$ with $\{l_i\} \subseteq \{l_j\}$ and $\Gamma \vdash t_i : T_i$ for all cases in which $i = j$.
4. If v is a value of type `List T`, then v is either an empty list `nil[T]` or of the form `cons v1 ... nil[T1]` with $T_1 <: T$ and $\Gamma, \vdash v_1 : T$.
5. If v is a value of type $T \rightarrow T'$, then v is a λ -abstraction `$\lambda x : T_1 . t_2$` with $T_1 <: T$ and $\Gamma, x : T_1 \vdash t_2 : T'$.

Given Lemma 4, we can show that a well-typed term is either a value or it can take a step.

Theorem 14 (Progress). *Let t be a closed, well-typed term. If t is not a value, then there exists a term t' such that $t \rightarrow t'$. If $\Gamma \vdash t : T$, then t is either a value, a term containing the forms `head nil[T]` or `tail nil[T]`, or there is some t' with $t \rightarrow t'$.*

Proof. By induction on the derivation of $\Gamma \vdash t : T$.

T-VAR Impossible since we are only looking at closed terms.

T-TRUE Immediate, since `true` is a value.

Appendix A. Soundness of λ_{Full}

T-FALSE Immediate, since `false` is a value.

T-IF $t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3, \quad \Gamma \vdash t_1 : \text{Bool}.$

By hypothesis, t_1 is a value or it can take a step. If it can take a step, rule E-IF applies. If it is a value, then by Lemma 4, either $t_1 = \text{true}$ or $t_1 = \text{false}$. In this case, either rules E-IFTRUE or E-IFFALSE apply.

T-ZERO Immediate, since `0` is a value.

T-SUCC $t = \text{succ } t_1, \quad \Gamma \vdash t_1 : \text{Nat}.$

By hypothesis, t_1 is a value or it can take a step. If it can take a step, rule E-SUCC applies. If it is a value, then by Lemma 4, $t_1 = nv$. The term `succ nv` is also a value.

T-PRED $t = \text{pred } t_1, \quad \Gamma \vdash t_1 : \text{Nat}.$

By hypothesis, t_1 is a value or it can take a step. If it can take a step, rule E-PRED applies. If it is a value, then by Lemma 4, $t_1 = \text{succ } nv$ or $t_1 = 0$. In those cases, rules E-PREDSUCC or E-PREDZERO apply.

T-ISZERO $t = \text{iszero } t_1, \quad \Gamma \vdash t_1 : \text{Nat}.$

By hypothesis, t_1 is a value or it can take a step. If it can take a step, rule E-ISZERO applies. If it is a value, then by Lemma 4, $t_1 = \text{succ } nv$ or $t_1 = 0$. Either rule E-ISZEROSUCC or rule E-ISZEROZERO apply.

T-APP $t = t_1 t_2, \quad \Gamma \vdash t_1 : T_{11} \rightarrow T_{12}, \quad \Gamma \vdash t_2 : T_{11}.$

By hypothesis, t_1 and t_2 are either values or they can take a step. If they can take a step, rules E-APP1 or E-APP2 apply. If both are values, then by the canonical forms lemma (Lemma 4), $t_1 = \lambda x : T_{11}. t_{11}$ and rule E-APPABS applies.

T-APP $t = \text{let } x = t_1 \text{ in } t_2, \quad \Gamma \vdash t_1 : T_1, \quad (\Gamma, x : T_1) \vdash t_2 : T_2.$

By hypothesis, t_1 is either a value or it can make a step. If it can, then rule E-LET applies. If it is a value, then rule E-LETV applies.

T-FIX $t = \text{fix } t_1, \quad \Gamma \vdash t : T_1, \quad \Gamma \vdash t_1 : T_1 \rightarrow T_1.$

By induction hypothesis, t_1 is either a value or it can take a step. If it can take a step, rule E-FIX applies. If its a value, by the canonical forms lemma (Lemma 4), $t_1 = \lambda x : T_1. t_2$. Therefore, rule E-FIX21 applies.

T-NIL Immediate, since `nil[T]` is a value.

T-CONS $t = \text{cons } t_1 t_2, \quad S, \Gamma \vdash t_1 : T_1, \quad \Gamma \vdash t_2 : \text{List } T_1.$

By hypothesis, t_1 and t_2 are either values or they can take a step. If they can take a step, then rules E-CONS1 or E-CONS2 apply. If both t_1 and t_2 are values, then t is also a value.

T-ABS Immediate, since $\lambda x:T . t_1$ is value.

T-ISNIL $t = \text{isNil } t_1, \quad \Gamma \vdash t_1 : \text{List } T_1.$

By hypothesis, t_1 is a value or it can take a step. If it can take a step, then rule E-ISNIL-3 applies. If it is a value, then by Lemma 4, $t = \text{nil}[T]$ or $t = \text{cons } v_1 \dots$. Then either rule E-ISNIL1 or E-ISNIL2 apply.

T-HEAD $t = \text{head } t_1, \quad \Gamma \vdash t_1 : \text{List } T_1.$

By hypothesis, t_1 is either a value or it can take a step. If it can take a step, rule E-HEAD applies. If it is a value, then by Lemma 4, either $t = \text{nil}[T]$ or $t = \text{cons } v_1 \dots$. Then either rule E-HEAD1 applies or the term is in the accepted normal form $t = \text{head nil}[T]$.

T-TAIL $t = \text{tail } t_1, \quad \Gamma \vdash t_1 : \text{List } T_1.$ By hypothesis, t_1 is either a value or it can take a step. If it can take a step, then rule E-TAIL applies. If it is a value, then by Lemma 4, either $t = \text{nil}$ or $t = \text{cons } v \dots$. Then either rule E-TAILV applies or the term is in the accepted normal form $t = \text{tail nil}[T]$.

T-RCD $t = \{l_i : T_i \mid i \in 1 \dots n\}, \quad \text{for each } i : \Gamma \vdash t_i : T_i.$

By induction hypothesis, each t_i is either a value or it can take a step. If one can take a step, then rule E-RCD applies. If each t_i is a value, then t is also a value.

T-PROJ $t = t_1.l_i, \quad \Gamma \vdash t_1 : \{l_i : T_i \mid i \in 1 \dots n\}.$

By hypothesis, t_1 is either a value or it can take a step. If it can take a step, then rule E-PROJ. If it is a value, then by Lemma 4, then $t = \{l_i : v_i \mid i \in 1 \dots n\}$ and rule E-PROJRCD applies.

T-SUB Results follow from induction hypothesis.

□

For proving preservation, we require an additional lemma that says that substitution preserves the type.

Lemma 5 (Substitution). *If $(\Gamma, x : T_2) \vdash t : T_1$ and $\Gamma \vdash t_2 : T_2$, then $\Gamma \vdash [x \mapsto t_2]t_1 : T_1$. Substitution therefore preserves the type [81].*

We can now show that if a term takes a step by the evaluation rules, its type is preserved.

Theorem 15 (Preservation). *Let t be a term and T a type. If $\Gamma \vdash t : T$ and $t \rightarrow t'$, then $\Gamma \vdash t' : T$.*

Proof. By induction of the derivation of $\Gamma \vdash t : T$.

T-VAR Cannot happen as we are only looking at closed terms.

T-TRUE Vacuously fulfilled, since `true` is a value.

Appendix A. Soundness of λ_{Full}

T-FALSE Vacuously fulfilled, since `false` is a value.

T-IF $t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3$, $\Gamma \vdash t_1 : \text{Bool}$, $\Gamma \vdash t : T$.

There are three rules by which t' can be derived: E-IF, E-IFTRUE and E-IFFALSE.

1. $t' = \text{if } t'_1 \text{ then } t_2 \text{ else } t_3$. By hypothesis, $t_1 \longrightarrow t'_1$ preserves the type. Therefore, by rule T-IF, $t : T$.
2. $t_1 = \text{true}$, $t' = t_2$. By rule T-IF, $t_2 : T$.
3. $t_1 = \text{false}$, $t' = t_3$. Same as second case.

T-ZERO Vacuously satisfied since `0` is a value.

T-SUCC $t = \text{succ } t_1$, $\Gamma \vdash t_1 : \text{Nat}$, $\Gamma \vdash t : \text{Nat}$.

There is only one rule E-SUCC by which $t' = \text{succ } t'_1$ can be derived. By hypothesis, $t_1 \longrightarrow t'_1$ preserves the type. Therefore, rule T-SUCC applies again and $t' : \text{Nat}$ preserves the type.

T-PRED $t = \text{pred } t_1$, $\Gamma \vdash t_1 : \text{Nat}$, $\Gamma \vdash t : \text{Nat}$.

There are two rules by which t' can be derived: E-PREDZERO and E-PREDSUCC.

1. $t_1 = 0$, $t' = 0$. By rule T-ZERO, $t' : \text{Nat}$.
2. $t_1 = \text{succ } nv_1$, $t' = nv_1$. By rule T-ZERO or T-SUCC, it must be that $t' : \text{Nat}$. The type is preserved.

T-ISZERO $t = \text{iszero } t_1$, $\Gamma \vdash t_1 : \text{Nat}$, $\Gamma \vdash t : \text{Bool}$.

There are three rules by which t' can be derived: E-ISZEROZERO, E-ISZEROSUCC and E-ISZERO.

1. $t_1 = 0$, $t' = \text{true}$. By rule T-TRUE, $t' : \text{Bool}$. The type is preserved.
2. $t_1 = \text{succ } nv_1$, $t' = \text{false}$. By rule T-FALSE, $t' : \text{Bool}$.
3. $t' = \text{iszero } t'_1$. By induction hypothesis, $t_1 \longrightarrow t'_1$ preserves the type. Therefore, by rule T-ISZERO, $t' : \text{Bool}$.

T-ABS Vacuously fulfilled, since $\lambda x:T. t_1$ is value.

T-APP $t = t_1 t_2$, $\Gamma \vdash t_1 : T_{11} \rightarrow T_{12}$, $\Gamma \vdash t_2 : T_{11}, S_2$, $\Gamma \vdash t_1 t_2 : T_{12}$.

There are three rules by which t' can be derived: E-APP1, E-APP2 and E-APPABS.

1. $t' = t'_1 t_2$ By induction hypothesis, $t_1 \longrightarrow t'_1$ preserves the type. Therefore, by rule T-APP, $t' : T_{12}$.
2. $t' = v_1 t'_2$. Same as first case.
3. $t' = [x \mapsto v_2]t_{12}$. By Lemma 5, substitution preserves the type. Therefore $t' : T_{12}$.

T-LET $t = \text{let } x=t_1 \text{ in } t_2, \quad \Gamma \vdash t_1 : T_1, \quad (\Gamma, x : T_1) \vdash t_2 : T_2, \quad \Gamma \vdash t : T_2.$

There are two ways t can be reduced: E-LET and E-LETV.

1. $t' = \text{let } x=t'_1 \text{ in } t_2.$ By induction hypothesis, $t_1 \rightarrow t'_1$ preserves the type. Then by rule T-LET, $t' : T_2.$
2. $t' = [x \mapsto v_1]t_2.$ By Lemma 5, the type is preserved, therefore $t' : T_2.$

T-FIX $t = \text{fix } t_1, \quad \Gamma \vdash t_1 : T_1 \rightarrow T_1, \quad \Gamma \vdash t : T_1.$

There are two rules by which t can be reduced: E-FIX1 and E-FIX2.

1. $t' = [x \mapsto \text{fix } (\lambda x : T_1 . t_2)]t_2.$ By Lemma 5, the type is preserved, therefore $t' : T_1.$
2. $t' = \text{fix } t'_1.$ By induction hypothesis, $t_1 \rightarrow t'_1$ preserves the type. Then, by T-FIX, $t' : T_1.$

T-RCD $t = \{l_i = t_i \mid i \in 1 \dots n\}, \quad \text{for each } i : \Gamma \vdash t_i : T_i, \quad \Gamma \vdash t : \{l_i : T_i \mid i \in 1 \dots n\}.$

t' can only be derived by rule E-RCD in which $t_i \rightarrow t'_i.$ By hypothesis, this preserves the type.

T-RCDPROJ $t = t_1.l_j, \quad \Gamma \vdash t_1 : \{l_i : T_i \mid i \in 1 \dots n\}, \quad \Gamma \vdash t : T_j.$

There are two rules by which t' can be derived: E-PROJ and E-PROJRCD.

1. $t = \{l_i = v_i \mid i \in 1 \dots n\}, t' = v_j.$ Due to rule T-RCD and T-PROJRCD, v_j must have type $T_j.$ Therefore, $t' : T_j.$
2. $t' = t'_1.l_j.$ By hypothesis, $t_1 \rightarrow t'_1$ preserves the type. Therefore, $t' : T_j.$

T-NIL Vacuously fulfilled, since $\text{nil}[T]$ is a value.

T-CONS $t = \text{cons } t_1 t_2, \quad \Gamma \vdash t_1 : T_1, \quad \Gamma \vdash t_2 : \text{List } T_1, \quad \Gamma \vdash t : \text{List } T_1.$ There are two rules by which t' can be derived: E-CONS1 and E-CONS2.

1. $t' = \text{cons } t'_1 t_2.$ By hypothesis, $t_1 \rightarrow t'_1$ preserves the type. Therefore, by rule T-CONS, $t' : \text{List } T_1.$
2. $t' = \text{cons } v_1 t'_2.$ Same as first case.

T-ISNIL $t = \text{isNil } t_1, \quad \Gamma \vdash t_1 : \text{List } T_1, \quad \Gamma \vdash t : \text{Bool}.$

There are three rules by which t' can be derived: E-ISNIL1, E-ISNIL2 and E-ISNIL3.

- $t' = \text{true}.$ By rule T-TRUE, $t' : \text{Bool}.$
- $t' = \text{false}.$ By rule T-FASE, $t' : \text{Bool}.$
- $t' = \text{isNil } t'_1.$ By hypothesis, $t_1 \rightarrow t'_1$ preserves the type. Therefore, by rule T-ISNIL, $t' : \text{Bool}.$

T-HEAD $t = \text{head } t_1, \quad \Gamma \vdash t_1 : \text{List } T_1, \quad \Gamma \vdash t : T_1.$

There are two rules by which t' can be derived: E-HEAD and E-HEADV.

Appendix A. Soundness of λ_{Full}

1. $t_1 = \mathbf{cons} \ v_1 \ v_2, t' = v_1$. By rule T-CONS, v_1 must be of type T_1 . Therefore, $t' : T_1$.
2. $t' = \mathbf{head} \ t'_1$. By hypothesis, $t_1 \longrightarrow t'_1$ preserves the type. Therefore, by rule T-HEAD, $t' : T_1$.

T-TAIL $t = \mathbf{tail} \ t_1, \quad \Gamma \vdash t_1 : \text{List } T_1, \quad \Gamma \vdash \mathbf{tail} \ t_1 : \text{List } T_1$.

There are two rules by which t' can be derived: E-TAIL and E-TAILV.

1. $t_1 = \mathbf{cons} \ v_1 \ v_2, t' = v_2$. Due to rules T-CONS, v_2 must have type $\text{List } T_1$. Therefore, $t' : \text{List } T_1$.
2. $t' = \mathbf{tail} \ t'_1$. By hypothesis, $t_1 \longrightarrow t'_1$ preserves the type. Therefore, by rule T-TAIL, $t' : \text{List } T_1$.

T-SUB Results follows from induction hypothesis.

□

As a direct consequence of Theorem 14 and Theorem 15, λ_{Full} is sound. A well typed term cannot get stuck unless it contains the form $\mathbf{head} \ \mathbf{nil}[T]$ or $\mathbf{tail} \ \mathbf{nil}[T]$.

List of Figures

1.1.	Stylized depiction of a knowledge graph.	1
1.2.	Example that combines the data graphs of two Web sites with an ontology	2
1.3.	SHACL shape that enforces that every instance of Person has a $\text{-hasName} \rightarrow$ relation.	4
2.1.	Syntax and evaluation rules for arithmetic expressions (NB).	10
2.2.	Typing rules for arithmetic expressions (NB).	13
2.3.	An example of an RDF graph G_1 (left) and its representation as a set of triples (right).	17
2.4.	RDF graph G_1 serialized using N3 Notation (namespace definitions are omitted).	18
2.5.	Syntax and Evaluation rules of conjunctive queries (CQs).	19
2.6.	SPARQL query in both abstract syntax (left) and standard SPARQL syntax (right).	20
2.7.	Syntax and Semantics of roles r and concept expressions C, D in the description logic \mathcal{ALCOIQ}	22
2.8.	Concept expressions derived from concept expressions as defined in Figure 2.7.	22
2.9.	Syntax and Semantics of axioms in the description logic \mathcal{ALCOIQ}	23
2.10.	Example knowledge base K_1	24
2.11.	A small knowledge base K_2 (left) and its RDF data graph (implicit knowledge in dashed lines).	26
2.12.	A small knowledge base K_3 (left) and its RDF data graph (implicit knowledge in dashed lines).	27
2.13.	An example of an RDF data graph G_2	28
2.14.	Example of a SHACL shape graph S_1 in N3 Notation.	28
2.15.	Validation report highlighting erroneous area in the graph G_2 (missing nodes and edges in dashed lines).	29
2.16.	Syntax and Evaluation rules of SHACL constraints.	30
2.17.	An example of a problematic, recursive constraint definition and RDF graph G_{rec}	31
2.18.	General form of target node queries in SHACL.	31
2.19.	RDF graph that is conformant to S_1 shown through the faithful assignment σ_1	33
2.20.	RDF data graph G_3 that conforms to the set of shapes S_2 even though no nodes conform to any shapes.	33
3.1.	Simply typed λ -calculus (λ_{\rightarrow}) with booleans, numerical values and let bindings.	37
3.2.	Typing rules for the simply typed λ -calculus (λ_{\rightarrow}).	40

List of Figures

3.3.	Simply typed λ -calculus with subtyping ($\lambda_{<}$).	43
3.4.	Fixpoint operator for recursion.	46
3.5.	Rules for a λ -calculus enriched with records.	48
3.6.	Rules for lists in the λ -calculus.	52
4.1.	Example knowledge base K_4 .	57
4.2.	Rules for assigning concepts to variables in queries.	58
4.3.	Syntax, Semantics and type system rules for λ_{DL} .	61
4.4.	Syntax and Evaluation rules of the typecase expression.	66
5.1.	Example SHACL shape graph.	74
5.2.	Examples for graphs that conform to the shape graph in Figure 5.2.	75
5.3.	Rules for assigning shapes to variables in queries.	76
5.4.	Syntax, Semantics and type system rules for λ_{SHACL} .	78
5.5.	Basic example for multiple faithful assignments.	87
6.1.	Counterexample that proves that OnlineUniversityShape is not contained in UniversityShape.	92
6.2.	Syntax and Semantics of properties p and concept expressions C, D in the description logic \mathcal{ALCOQ} .	103

List of Algorithms

1.	Definition of TYPEOF for λ_{\rightarrow} .	42
2.	Definition of TYPEOF and SUBTYPE for $\lambda_{<}$.	44
3.	Definition of LUB and GLB for $\lambda_{<}$.	45
4.	Definition of TYPEOF and SUBTYPE for $\lambda_{Records}$.	50
5.	Definition of LUB and GLB for $\lambda_{Records}$.	51
6.	Definition of TYPEOF and SUBTYPE for λ_{Full} .	54
7.	Definition of LUB and GLB for λ_{Full} .	54
8.	Definition of TYPEOF and SUBTYPE for λ_{DL} .	64
9.	Definition of LUB and GLB for λ_{DL} .	64
10.	Definition of TYPEOF for λ_{SHACL} .	81
11.	Definition of SUBTYPE for λ_{SHACL} .	82
12.	Definition of LUB and GLB for λ_{SHACL} .	82
13.	Definition of ELABORATE for λ_{SHACL} .	83
14.	Improved definition of SUBTYPE for λ_{SHACL} .	105
15.	Improved definition of LUB for λ_{SHACL} .	105

MARTIN GERHARD LEINBERGER

Institute for Web Science and Technologies, University of Koblenz-Landau
Universitätsstr. 1, 56073 Koblenz

EDUCATION AND QUALIFICATION

University of Koblenz-Landau, Koblenz Master of Science in Computer Science	<i>2011 - 2013</i>
Johannes Gutenberg University, Mainz Bachelor of Science in Computer Science	<i>2007 - 2011</i>

POSITIONS

Scientific Assistant at Institute for Web Science and Technologies University of Koblenz-Landau, Koblenz	<i>Nov 2013 - Present</i>
Intern at IBM Germany, Research and Development GmbH Mainz	<i>Aug 2009- Oct 2009</i>

RESEARCH

My research focus lies on programming with semantic data. In particular, I look into type checking of programs working with semantic data in order to verify the absence of run-time errors. Additionally, I investigate the integration of semantic data in programming in general, such as programming language integrated querying and autocompletion mechanisms.

TEACHING

Algorithms and Data Structures, Lecture	<i>2020 - 2021</i>
Big Data, Lecture	<i>2020</i>
Algorithms and Data Structures, Tutorial	<i>2014 - 2019</i>
Artificial Intelligence, Tutorial	<i>2015 - 2019</i>
Research Practical on AI approaches for real-time strategy games	<i>2018</i>
Proseminar on Graph Algorithms	<i>2015</i>

REVIEWING ACTIVITIES

Int. Symposium on Principles and Practice of Declarative Programming (PPDP)	<i>2019</i>
Int. Conference on Software Language Engineering (SLE)	<i>2018</i>
IEEE Internet Computing	<i>2015</i>

PUBLICATIONS

1. Favre J.-M., Lämmel R., Leinberger M., Schmorleiz T., and Varanovich A (2012). Linking Documentation and Source Code in a Software Chrestomathy.
In: *Working Conference on Reverse Engineering (WCRE)*. IEEE Computer Society, pp.335–344.
2. Lämmel, R., Leinberger M., Schmorleiz T., and Varanovich A (2014). Comparison of feature implementations across languages, technologies, and styles.
In: *Proceedings IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering*. IEEE Computer Society, pp.333–337.

3. Lämmel, R., Varanovich A., Leinberger M., Schmorleiz T., and Favre J.-M. Declarative Software Development (2014): Distilled Tutorial.
In: *Proceedings of the International Symposium on Principles and Practice of Declarative Programming (PPDP)*. ACM, pp.1–6.
4. Leinberger, M., Scheglmann, S., Lämmel, R., Staab, S., Thimm, M., Viegas, E.: Semantic Web Application Development with LITEQ.
In: *Proceedings of the 13th International Semantic Web Conference (ISWC 2014)*. LNCS, vol. 8797, pp. 212–227. Springer (2014).
5. Scheglmann, S., Lämmel, R., Leinberger, M., Staab, S., Thimm, M., Viegas, E.: IDE Integrated RDF Exploration, Access and RDF-Based Code Typing with LITEQ.
In: *The Semantic Web: ESWC 2014 Satellite Events (ESWC 2014)*. LNCS, vol. 8798, pp. 505–510. Springer (2014).
6. Scheglmann, S., Leinberger, M., Lämmel, R., Staab, S., Thimm, M.: Property-based typing with LITEQ.
In: *Proceedings of the 13th International Semantic Web Conference (ISWC 2014), Posters & Demonstrations Track*. CEUR Workshop Proceedings, vol. 1272, pp. 149–152. CEUR-WS.org (2014)
7. Staab, S., S. Scheglmann, M. Leinberger, and T. Gottron (2014). Programming the Semantic Web.
In: *Proceedings The Semantic Web: Trends and Challenges - 11th International Conference (ESWC)*. Vol. 8465. LNCS. Springer, pp.1–5.
8. Scheglmann, S., M. Leinberger, T. Gottron, S. Staab, and R. Lämmel (2016). SEPAL: Schema Enhanced Programming for Linked Data.
KI **30**(2), 189–192.
9. Hartenfels, C., M. Leinberger, R. Lämmel, S. Staab: Type-Safe Programming with OWL in Semantics4J.
In: *Proceedings of the ISWC 2017 Posters & Demonstrations and Industry Tracks, 16th International Semantic Web Conference (ISWC 2017)*. CEUR Workshop Proceedings, vol. 1963.
10. Leinberger, M., Lämmel, R., Staab, S.: The Essence of Functional Programming on Semantic Data.
In: *Proceedings of the 26th European Symposium on Programming (ESOP 2017)*. LNCS, vol. 10201, pp. 750–776. Springer (2017).
11. Leinberger, M., P. Seifer, C. Schon, R. Lämmel, S. Staab: Type Checking Program Code Using SHACL.
In: *Proceedings of the 18th International Semantic Web Conference (ISWC 2019)*. LNCS, vol. 11778, pp. 399–417. Springer (2019).
12. Seifer, P., J. Härtel, M. Leinberger, R. Lämmel, and S. Staab (2019). Empirical study on the usage of graph query languages in open source Java projects.
In: *Proceedings of the International Conference on Software Language Engineering (SLE)*. ACM, pp.152–166.
13. Seifer, P., M. Leinberger, R. Lämmel, S. Staab: Semantic Query Integration With Reason.
Programming Journal **3**(3), 13 (2019).
14. Leinberger, M., P. Seifer, T. Rienstra, R. Lämmel, S. Staab: Deciding SHACL Shape Containment Through Description Logics Reasoning.
In: *Proceedings of the 19th International Semantic Web Conference (ISWC 2020)*. LNCS, vol. 12506, pp. 366–383. Springer (2020).